PROGRAM ANALYSIS BY DIGITAL COMPUTER

by

DANIEL UNDERWOOD WILDE

B.S.E.E., University of Illinois
1961

S.M., Massachusetts Institute of Technology
1962

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June, 1966

Signature of Author Daniel U. Wilde
Department of Electrical Engineering, May 13, 1966

Certified by______________________________
Thesis Supervisor

Accepted by______________________________
Chairman, Department Committee of Graduate Students

PROGRAM ANALYSIS BY DIGITAL COMPUTER

by

DANIEL UNDERWOOD WILDE

Submitted to the Department of Electrical Engineering on May 13, 1966 in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

ABSTRACT

A comparison of the properties of non-modifying and self-modifying programs leads to the definition of independent and dependent instructions. Because non-modifying programs contain only independent instructions, such programs can be analyzed by a straight forward, two-step analysis procedure. First, the program control flow is detected; second, that control flow is used to determine the program data flow or data processing. However, self-modifying programs can also contain dependent instructions, and the program control flows and data flows exhibit cyclic interaction. This cyclic interaction suggests the use of an iterative or a relaxation analysis technique. The initial step in the relaxation procedure determines a first approximation to control flow; the second step then finds a first approximation to data flow. These two stêps are repeated until a steady-state condition is reached.

Algorithms for implementing the first iteration are presented. These algorithms are capable of analyzing programs which modify their control and processing instructions during the course of execution. In addition, data structures are described which permit the construction of functional expressions for the data flow or information processing. Finally, actual output flowcharts of self-modifying programs are displayed.

Thesis Supervisor: Herbert M. Teager
Title: Associate Professor of Electrical Engineering

ACKNOWLEDGEMENT

The author would like to express his deepest appreciation to Professor Herbert M. Teager who not only served as thesis supervisor, but also was a source of inspiration, a friend, and a confidant.

Thanks are also due the thesis readers, Professors Donald C. Carroll and Thomas G. Stockham, for their constructive criticism and evaluation of the thesis research.

The author would also like to thank his friends and associates at Project MAC for their interest and comments during many discussions on the thesis subject. Particular thanks go to A. Scherr, R. Thurber, and O. Wright. In addition the author would like to thank the administration and staff of Project MAC for the support and use of the time-shared system.

Finally, the author would like to express his heartfelt thanks to his parents for their continuing support; and to his wife, Marylin, for her unending confidence and encouragement.

TABLE OF CONTENTS

CHAPTER 1

SUMMARY

This chapter outlines the organization of this thesis.

The second chapter is an introduction to automatic program analysis by digital computer. Automatic program analysis is defined as the construction of a flowchart from an original source program without human assistance. Development of such an analysis capability is motivated by its possible use as a documentation and debugging tool. The history of automatic program analysis is presented. The purposes, objectives, scope, and restrictions of the thesis are stated.

The third chapter presents the major problems of analyzing programs which modify themselves. A comparison of the properties of non-modifying and self-modifying programs leads to a statement of the general analysis problem and a general analysis procedure.

The fourth chapter discusses the major techniques used in the general analysis procedure. The solution philosophy required for a successful analysis is stated. The general organization of the analysis system is outlined. Finally, a more detailed description of the individual analysis techniques is given.

The fifth chapter displays the results of applying the existing analysis system to example programs. The layout and symbols of the output flowcharts are explained. Automatically produced flowcharts of programs containing particular analysis problems are presented.

The sixth chapter summarizes and evaluates the specific results shown in the earlier chapters and discusses reasonable extensions of these results.

The first appendix contains the general flowcharts of the analysis system subroutines. The second appendix displays output flowcharts produced by applying the analysis system to some of its own subroutines.

CHAPTER 2

INTRODUCTION

This chapter is an introduction to automatic program analysis. First, the general problem of such analysis is presented, and includes a discussion of what automatic program analysis involves and why it is useful. Finally, the purpose, objectives, scope, and restrictions of this thesis are given.

2.1 MOTIVATION

In the early days of computer development, a detailed step-by-step machine-language program, i.e. numerical code, had to be written before a computer could be used to solve any problem. Because writing each new program in machine-language required excessive coding and debugging time, special programming aids were devised. Today, all machines have assemblers that permit the programmer to use symbolic operation codes and symbolic addresses. In addition, debugging packages and memory-dump routines help the program tester reduce debugging and testing time. Finally, general-purpose languages, such as FORTRAN and MAD, enable inexperienced programmers to write programs without worrying about machine-language errors.

All of these programming aids are designed to help the programmer write a new routine, but are of restricted use in understanding or modifying an existing program even by its original author. In such a situation there is no substitute for adequate, clear, and pedagogically meaningful documentation of the intent and details of the programming algorithms. In the absence of such information, a user would struggle through the code to convert the existing program back into a block diagram or a flowchart. After the flowchart was reconstructed, the programmer could begin to understand both the function and algorithms of the routine as the sum of its parts. During such a reconstruction, a human programmer performs many tasks which could be automated; and thus, major portions of such automatic analysis could be performed by the computer.

Automatic program analysis can clearly be applied to any aspect of producing pedagogically meaningful program documentation. For our purposes, we shall consider the construction of an accurate and concise flowchart from an original assembly-language source program without human assistance to represent a useful form of such information. This flowcharting procedure must produce the flowchart "boxes" with their sequential processes, and all such procedures must be interconnected. The flowchart boxes and interconnections represent the control flow of the program, i.e. the program instruction execution sequence. The functional relationships inside the flowchart boxes express the data flow of the program, i.e. the program information processing. Flowcharts are generally accepted as the sine qua non of documentation procedures. The major difficulties in machine generated flowcharts (over and above the sheer difficulty of the problem) are no different from those encountered in hand generated ones. The more compact, concise, and meaningful the document, the greater the departure from machine and processing detail; and thus the more reasoning and abstraction required of the "analyst" and less of the user. Results of this automatic analysis even in a somewhat detailed form would be useful either as a debugging tool or as a documentation tool.

As a debugging tool, the analysis program could analyze and display all possible execution paths, not just those that might be executed during the testing session. At the same time, the analysis program could call attention to any obvious program inconsistencies, before the debugging and testing sessions began.

As a documentation tool, the analysis program could automatically provide final flowcharts for program documentation. This would allow the programmer to spend more of his time generating program code and less time documenting code. If flowcharts were prepared automatically, it would be easy to have an up-to-date version immediately after code corrections or additions were made. Also, a current flowchart would help reduce coding interruptions due to programming staff changes. If the results of automatic analysis were presented in a standardized mathematical form, it should be possible for a non-programmer with a general mathematical background to understand the algorithm and comprehend its implications. Finally, automatic program analysis should increase the human capability for understanding large programmed systems, by raising the level at which the human being assumes an analytic role.

Besides the direct use of program analysis for debugging and documentation, there are problems which can build on the results of such analysis. The solution of these problems requires an understanding of the interaction between programming languages and the execution of their generated machine code. Examples of three such problems are given, and the following discussion includes a statement of the problem and justification for its solution.

The development of large, interactive digital systems has made the estimation of program execution time less reliable (13). In a time-sharing system the operations manager cannot predict the throughput of his system, just as in a large military command-and-control system the commander cannot ascertain the information input conditions which will saturate his facility. A better understanding of the relationship between a programmed system and its machine execution requires a knowledge of execution times and storage requirements as a function of the program. With such data, a system analyst can decide what improvements need to be made and what improvements can be made.

Today, it is still accepted that programs which are to be used repeatedly should be written in machine-language, while those used just now and then could be written in a general-purpose compiler language. Thus, it is possible to pay for higher programming costs with the savings from machine-time expenses. However, this balance can shift because of a shortage of assembly-language programmers. Since there has always been a shortage of capable programmers, why not develop an automatic machine-code-optimization procedure that could be used either during or after the compilation of a program (10). Thus, relatively efficient machine code could be generated by relatively inexperienced programmers.

The last example concerns the reprogramming effort required by a change of machines. At present, this usually means converting to a new language. However, future system managers will be concerned not only with changes in machine language, but also changes in machine structure (e.g., from single processing to multiprocessing). If the switch is to be worthwhile, a manager must take advantage of the new structure, and he is faced with an inevitable reprogramming task.

Also, the system manager would like to have his users or customers take advantage of his new facilities. However, at the same time he must not increase a user's cost per unit of processing. The answer to this problem is to provide an automatic reprogramming system which can convert from one language to another and still increase efficiency by taking advantage of all the new features which prompted the machine change (9).

Although hopefully a clear case has been made for the _desirability_ of machine program analysis, its feasability, practical utility, and difficulty of realization are far from clear. Utility assessment must await availability, and the problem is far from trivial. In fact it is the _impossibility_ of finding a complete, closed form solution to the problem of program analysis (a known consequence of Turing machine theory) that has in part impeded the needed theoretical interest in the problem. Such applied work as has been noted in the literature is scattered and is far short of the requirements for even a rudimentary flowcharter.

2.2 HISTORY

The purpose of this section is to review the literature that has appeared in the area of program analysis. The review is intended to show what has been done so that the context of this thesis may be seen. This presentation is divided into four parts: Directed-Graph Theory as Applied to Program Analysis; Program Analysis of Compiler-Language Source Programs; Program Analysis of Machine-Language Source Programs; and the Presentation of Program Analysis Results via Flowcharts. The work which we will describe is generally much too restrictive to be useful for the patterns of assembly-language coding which are generally utilized.

2.2.1 Directed-Graph Theory

A digital computer program can be represented by a directed-graph model, if all control paths are known ab initio. Nodes of the graph represent blocks of code, and branches of the graph represent control paths. With such a model, results of classical directed-graph theory can be applied to the program analysis problem, in the sense of predicting connectivity between arbitrary nodes.

R. T. Prosser (11), in work done in 1959, describes the analysis of directed graphs by the use of boolean matrices. Two boolean matrices are associated with each graph: the first is called the connectivity matrix, and contains the topological structure of the diagram; the second is called the precedence matrix, and contains the precedence relations of the graph.

The connectivity matrix is an n by n boolean matrix, $A = (a_{ij})$, where n is the number of program blocks and $a_{ij} = 1$ if program block j is just preceded by program block i. The precedence matrix is an n by n boolean matrix, $B_m = (b_{ij})$, which is derived from the connectivity matrix by performing elementary matrix computations on A exactly m times. Depending on the operations used, $b_{ij} = 1$ can indicate that it is possible to proceed from block i to block j in either exactly m steps or at most m steps.

C. V. Ramamoorthy (12), in work done in 1965, uses the connectivity matrix and precedence matrices to determine the structural characteristics of the program represented by the boolean matrices. He presents algorithms for detecting blocks which cannot be reached from the starting block; for finding which blocks are included in at least one loop; for partitioning a graph into its unconnected subgraphs; and for determining the entry and exit blocks. Obviously, these determinations are of only incidental interest in understanding a procedure or deriving its flowchart. For a general review of graph theory, see C. Berge (1).

2.2.2 Program Analysis of Compiler-Language Source Programs

L. Krider (8), in work done in 1964, describes an algebraic representation of the control flow of a computer program and presents an algorithm for manipulating such a representation into a form which could be used to draw a flowchart. The algorithm works on the assumption that the principal information about program flow is contained in its loop structure. The algorithm also requires that all possible destinations of all transfer instructions must be known in advance. Thus, this procedure can only be used on algebraic source-language programs. Such a "pattern of code" is far more restrictive than is utilized in assembly-language programming.

2.2.3 Program Analysis of Machine-Language Source Programs

L. M. Haibt (3), in work done in 1959, describes a program, the FLOWCHARTER, which automatically produces flowcharts of programs whose instructions are fixed and not modified or calculated during execution. The output of the FLOWCHARTER is a set of flowcharts showing various levels of detail, where each part of a chart is shown in more detail on a succeeding chart. The FLOWCHARTER is divided into four main parts: preprocessing, flow analysis, computation summary, and output.

The preprocessors transform input source language instructions into an internal language. This permits the FLOWCHARTER to handle different source languages by simply using the proper preprocessor. The flow analysis program determines what information goes on each flowchart level. This routine first determines individual blocks and then groups the smaller blocks together into larger blocks. The computation summary program determines, for each block, which cells are used in input/output, which cells are used in calculations, and which cells are calculated. No functional relationships are derived; only the variable names are listed. The output program prints the various flowcharts.

H. M. Teager, in an unpublished work, developed a cross-referencing program. The input of the program is a 709 FAP source-language program, while the output is a program listing plus cross-reference information. For each instruction location, the cross-reference information indicates the location of all instructions in the program that might effect the given instruction. For example, if an instruction changes or uses the contents of a cell, all locations which similarly modify or use that cell are listed beside the given instruction. Although helpful, sometimes the sheer volume of output makes the information useless.

2.2.4 Presentation of Program Analysis Results

G. Hain and K. Hain (4) have developed a program which will draw flowcharts. The blocks of the chart are positioned so that logically-close blocks are physically close, and there is a minimum number of connecting-line crossings. Likewise, W. Sutherland, in an unpublished work, used the SKETCHPAD program developed by I. Sutherland (15), to display flowcharts. In both of these works, output presentation was the major concern, and the necessary machine analysis was assumed to have been derived by other means.

2.3 PURPOSE AND SCOPE OF THIS THESIS

This paper has two purposes. The first is to present algorithms for analyzing programs which modify their control and processing instructions in the course of execution. Examples of such self-modification are computed changes in operation code or operand address of instructions. The second purpose of this paper is to present data structures which will permit a functional expression of program data or information processing. These algorithms and data structures were utilized in a program analysis system which produced data and control flowcharts from assembly-language code. Even though the procedures and data structures were developed for a specific computer and its assembly-language, the results are of general theoretic and practical interest. The machine incorporates all of the most sophisticated operations of any existing machine short of a true multiprocessor, and thus, there are no major "surprises" to be expected from minor perturbations in the common structure of forthcoming machines in the near future, whether more or less powerful.

The analysis and display procedures are general in scope; the concepts apply to all machines and all programs. For purposes of experimentation, the analysis and display algorithms were written for the IBM

7094 single-address machine (5) and the FAP assembler language (6). Input to the analysis program is the BCD listing produced by the FAP assembler. Output from the analysis program is a flowchart, where block interconnections show the program control flow and symbolic functional expressions inside the blocks show the program data or information flow. In addition, pertinent cross-reference information is given beside each block. This information permits a human user to begin analyzing the program at a more sophisticated level if the automatic procedures break down. Sufficient routines have been written to validate the proposed analysis algorithms and evaluate the results of the analysis programs.

Chapter 3

A DISCUSSION OF THE ANALYSIS OF SELF-MODIFYING PROGRAMS

The purpose of this chapter is to introduce the major problems of automatic analysis of self-modifying programs. First, a comparison of the properties of non-modifying and self-modifying programs with respect to data and control flow leads to a statement of the general analysis problem. Second, the general solution procedure of successive approximations utilized to solve this problem is outlined. Third, the problems introduced by the solution procedure are discussed. Finally, examples of self-modifying programs further illustrate the analysis problems. In the description to follow, moderate familiarity with assembly-language programming and the specific mnemonics and conventions of IBM's FAP will be assumed (5 and 6).

3.1 THE GENERAL ANALYSIS PROBLEM

Before the general analysis problem is stated, it would be good to review the special case of programs which do not modify themselves. This review describes the special property of non-modifying programs which permits a straight-forward, direct analysis procedure.

If a program is non-modifying, the set of all possible outcomes for each instruction is a function of the instruction itself and is independent of all other program instructions. For example, an absolute transfer instruction, TRA Y, is an independent instruction because all of its outcomes are determined by the instruction itself. On the other hand, a tagged transfer instruction, TRA Y, 1, is a dependent instruction because its outcomes are a function of the contents of the index register and thus the instructions and data which affected it. There is a wide class of such dependent instructions which must be treated in the general case.

The independence property of non-modifying programs permits a straight-forward, two-step analysis procedure. First, the program control flow is determined by finding the outcome sets of all the transfer or control instructions. These results are used to draw the flowchart box outlines and interconnections. Second, the program data flow is determined by finding the outcome sets of all the information processing instructions. These results are then processed as a function of the control flow to produce the symbolic functional expressions for inside the flowchart boxes. In summary, the independence property permits a two-step analysis procedure because the control flow can be found without regard to the data flow.

However, if a program is self-modifying, the above two-step analysis procedure cannot be used because it assumes instruction independence. If a program contains dependent instructions, such as a tagged transfer instruction, the control and data flows are a function of each other. The outcome set of a tagged transfer is a function of the index register loading instruction, but the set of index loading instructions can be a function of the outcomes of the tagged transfer instruction itself. Because of this control flow - data flow interaction, a new analysis procedure is needed for self-modifying programs. To be feasible, such a procedure must perforce fall short of a complete dynamic analysis of the program's execution, and instead consider just a few static iterations.

3.2 THE GENERAL ANALYSIS PROCEDURE

If the control flow and data flow of a self-modifying program are to be determined, a procedure must be found for handling the control flow - data flow interaction cycle. This cyclic behavior of self-modifying programs suggests the use of an iterative or a relaxation solution technique.

Since data flow is always a function of control flow, the initial step in the relaxation solution procedure should determine a first approximation to the control flow. The second step would then determine a first approximation to the data flow as a function of control flow. The first two steps would be repeated until all the outcomes of all the dependent instructions have been found and the analysis results have reached a steady-state condition. Only then can the control flow results be used to construct the flowchart box outlines and interconnections, and the data flow results to produce the symbolic functional expressions for inside the flowchart boxes.

3.3 THE RELAXATION SOLUTION PROBLEMS

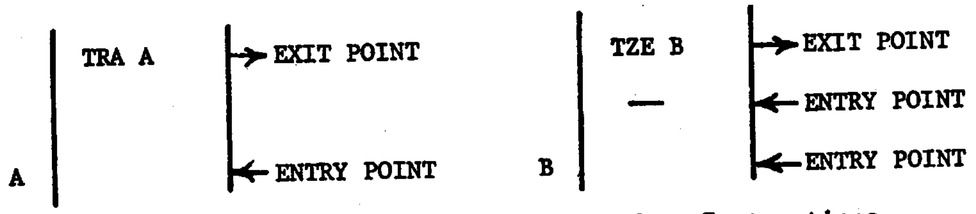
The relaxation solution procedure is the iterative application of the two-step analysis process for non-modifying programs. Because of the control flow - data flow interaction cycle of self-modifying programs, both steps must be modified. The purpose of this section is to review the problems solved by the two-step procedure and to show how this process must be modified to solve the relaxation problems.

3.3.1 Control Flow Modifications

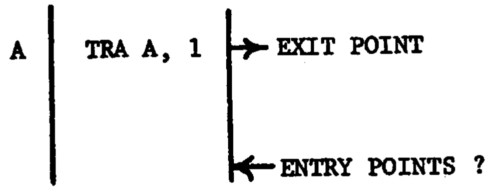
Control flow represents the program instruction execution sequence and is used to construct the flowchart box outlines and interconnections. This execution sequence can be modeled by a directed graph where nodes represent flowchart boxes and directed branches represent box interconnections. More specifically, let each node of the control graph represent a program block. Let a block be defined as a sequential set of instructions between a transfer entry point and the next transfer entry or exit point. Thus, a block is completely processed once its first member instruction is executed. Therefore, a directed graph whose nodes represent program blocks displays only execution sequence information. The major control flow graph construction problems are breaking the program into blocks and then interconnecting those blocks in proper sequence. Now, the differences between finding the control graph of a non-modifying program and of a self-modifying program are discussed.

The first control graph construction step is the detection of all control or transfer instructions. Each of these instructions generates a set of outcomes, i.e. entry and exit points. For non-modifying programs, all entry and exit points can be determined from the individual control instructions. Figure 3.1a shows examples of entry and exit points generated by independent control instructions. However, in the case of self-modifying programs, some entry and exit points cannot be immediately determined because of dependent instructions. Figure 3.1b shows an example of such a dependent instruction, the tagged transfer, where the entry points cannot be determined from the transfer instruction itself. Therefore, the control graph construction procedure must be modified to handle missing entry and exit points.

Figure 3.1 - Entry and Exit Points



a. Entry and Exit Points Generated by Independent Instructions



b. Entry and Exit Points Generated by a Dependent Instruction

In the second construction step the entry and exit points are processed to determine the program blocks. In the non-modifying case, the application of the block definition is straight forward. In the self-modifying case, some entry and exit points are initially missing. Therefore modification to the block definition is required so that a first approximation to the program blocks can be made.

The third construction step interconnects the blocks or nodes in the proper execution sequence. In the case of non-modifying programs, all interconnections can be made because all control instruction outcomes are known and blocks are completely defined. In the case of self-modifying programs, some block connections cannot be made because of incomplete control instruction outcome sets. Therefore, the block interconnection procedure must be modified so that assumed control graph branches can be inserted at points where incomplete outcomes occur.

The final construction step places the control flow information into some data structure. The control flow information of a non-modifying program can be stored in a rigid data structure because its information is completely known and is not changed by later analysis. However, the data structure used to represent the self-modifying program needs to be flexible because it contains information which might be updated by later analysis results.

3.3.2 Data Flow Modifications

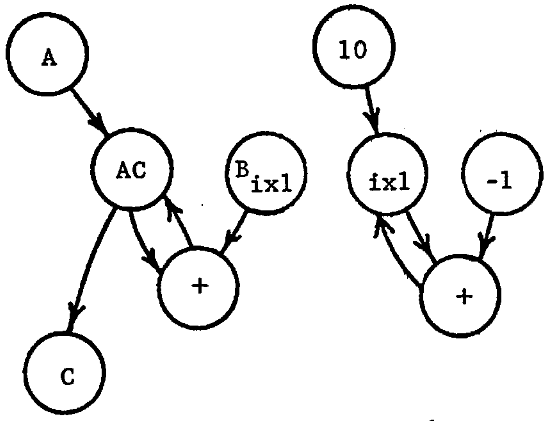
Data flow represents the data or information processing performed by the program and is used to generate the functional expressions for inside the flowchart boxes. This data processing can be modeled by a directed graph where the nodes represent cell references or operators and the directed branches represent the processing sequence. A cell is either a memory location or a central processor register. An operator is a machine operation, such as ADD or MULTIPLY.

The data flow graph removes the sequential constraint imposed by the digital computer. This removal permits a better presentation of the program's data processing algorithm by removing references to temporary storage and displaying parallel processing paths. The data flow is an implicit function of the control flow because control flow determines the order of instruction execution and thus the arrangement of data flow graph nodes and branches. Figure 3,2 shows a simplified program and its data graph.

Figure 3.2 - A Data Flow Graph

```
          AXT      10,1
          CLA      A •
REPEAT    ADD      B,1
          TIX      REPEAT,1,1
          STO      C
```

a. The Program



b. Its Data Flow Graph

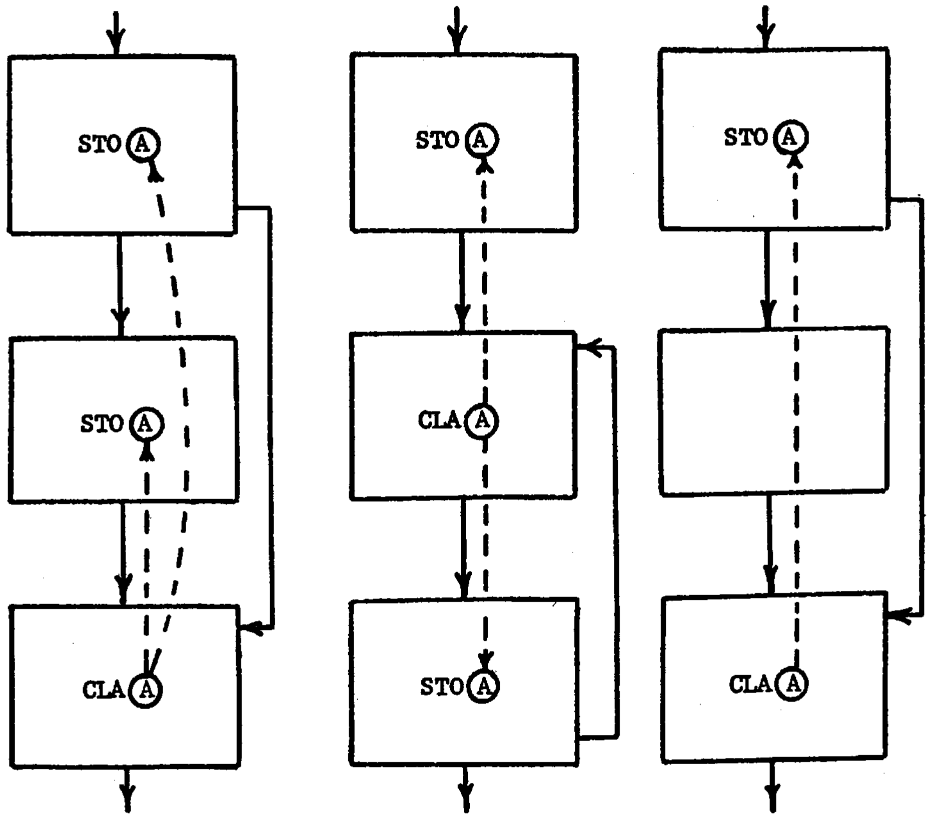
The major data flow graph construction problems are determining where and how each cell is referenced and then interconnecting those references in the proper sequence to form the data flow graph. Now, the differences between finding the data flow graph of a non-modifying program and of a self-modifying program are discussed.

The first data graph construction step is the detection of all instructions which change or use data or information. Each of these instructions generates a set of outcomes, i.e. a set of references to various cells. In the case of non-modifying programs, the reference outcomes of each instruction can be found from the instruction itself. While in the case of self-modifying programs, some outcomes may not initially be known. For example, the cells referenced by the dependent instruction, CLA **, cannot be determined until after the actual address of the instruction itself has been found. Thus, the reference detection procedure must be modified to handle dependent data referencing instructions.

The second construction step determines the effect of each cell reference. The reference effect can be found from the instruction itself. Let a reference which changes the contents of a cell be known as an active reference. Let a reference which only uses the contents of a cell be known as a passive reference. For example, the CLA A instruction makes a passive reference to A and then an active reference to the accumulator, AC. The ADD B instruction first makes a passive reference to cells B and AC and then makes an active reference to the AC.

The third construction step determines the processing sequence of the data references. When a program makes a passive reference to a cell, it obtains the contents placed there by that cell's latest executed active reference. In a static analysis it is only possible to find all possible latest active references for each passive reference; only a dynamic or interpretive process can detect the single latest active reference. The latest reference set for each passive reference can be

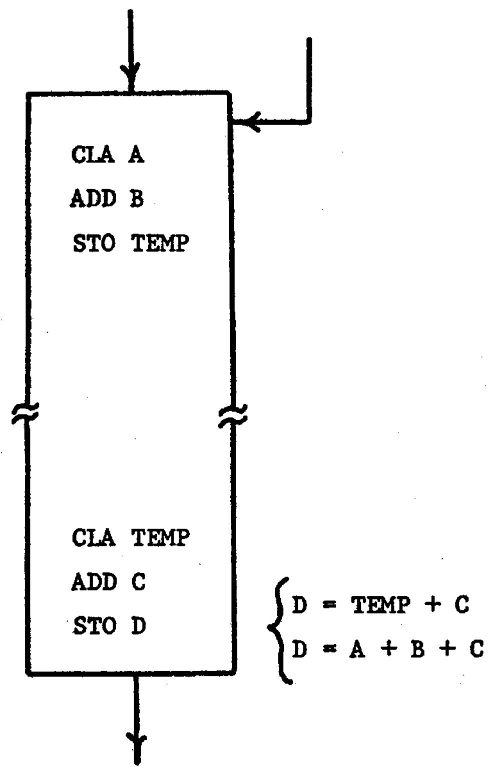
Figure 3.3 - Latest Reference Sets



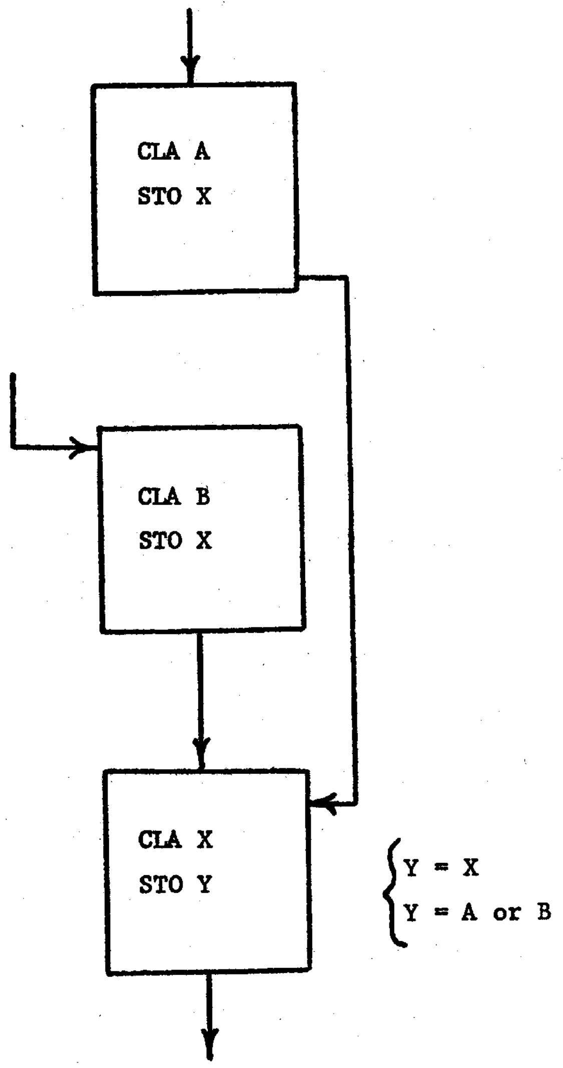
a. Dual Search Path   b. Loop Search Path   c. Parallel Search Path found by searching back through the program as a function of the control flow until all control paths are terminated by an active reference. Figure 3.3 shows examples of latest reference sets. The dashed arrows indicate latest references produced by passive reference - active reference matches. In the case of non-modifying programs, all data references are known and control flow is completely determined. Such is not the case for self-modifying programs. Since individual passive references can be missing, not all the latest reference sets may be found. Since individual active references can also be missing, latest reference searches may be improperly terminated. Finally, since control flow paths can be missing because they are functions of yet to be determined data flow, latest reference searches may be incorrect. Thus, the latest reference searching procedure must be modified to handle dependent instructions.

The final construction step places the latest reference information into a data structure which permits the generation of symbolic functional expressions for inside the flowchart boxes. The data structure must allow the analysis program to carry latest reference expressions forward to each passive reference that needs them. The data structure must also permit the analysis program to compress and simplify those functional expressions. Figure 3.4 shows examples of functional expressions. The second expression in each example is preferred. In the non-modifying program, all control paths and data references are known. Therefore, the latest reference structure can be rigid, and the functional

Figure 3.4 - Functional Expressions



a. Removal of Temporary Stores



b. Multiple Values

expressions are final. In the self-modifying program case, some control paths and data references can be missing. The latest reference data structure must be flexible because its information may be changed in later iterations.

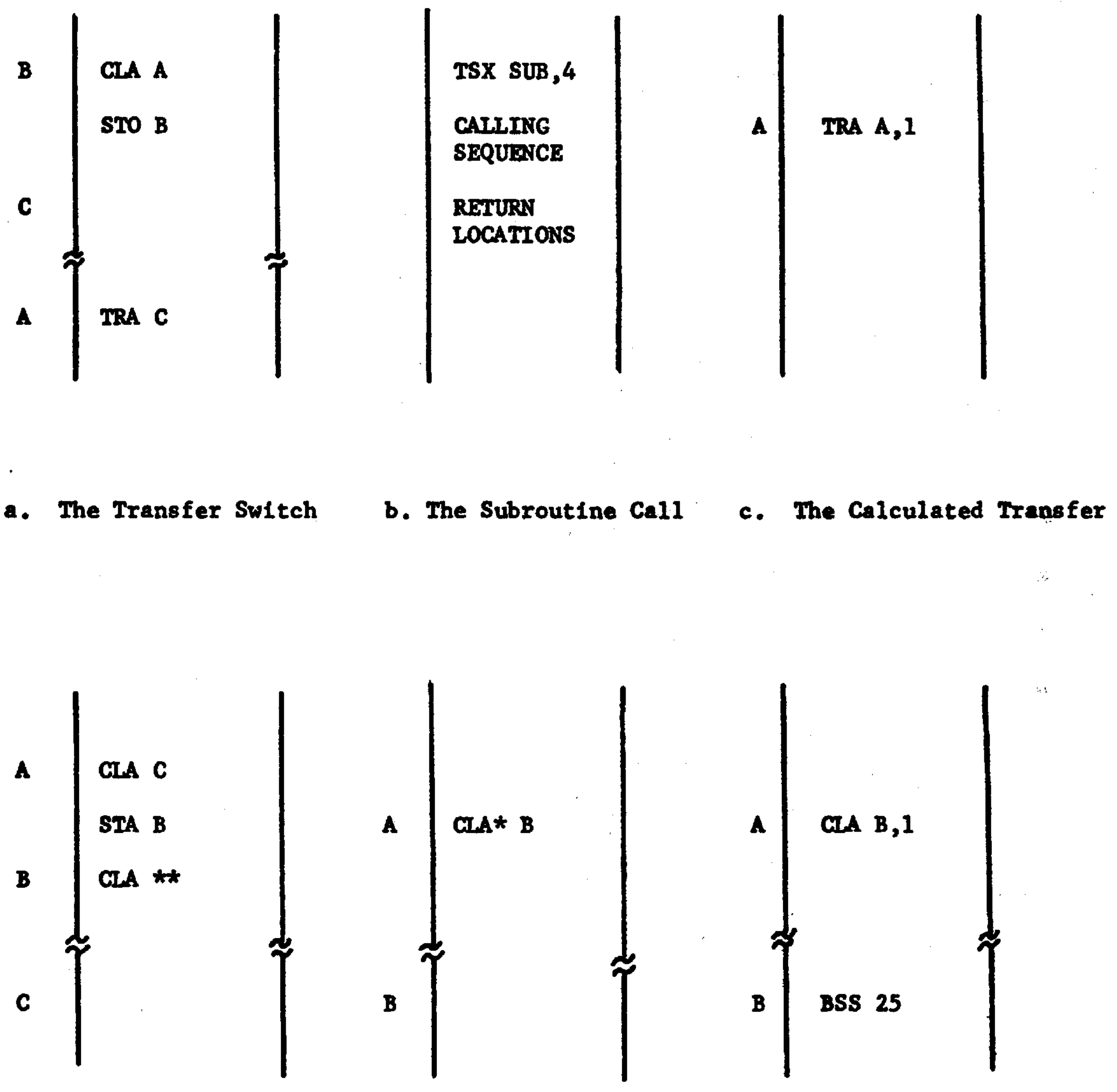
3.4 DEPENDENT INSTRUCTIONS

Because of the large number of machine instructions and assembly pseudo-operations in the FAP assembly-language, it is necessary to limit the number and format of dependent instructions which the automatic analysis program will initially handle. The purpose of this section is to list and describe these dependent instructions.

3.4.1 The Transfer Switch

The first example of a control flow - data flow interaction problem is the transfer switch. A transfer switch occurs when a program changes its execution path by replacing or modifying its own instructions. Figure 3.5a shows one of the many forms of the transfer switch. In this example, the transfer instruction at location A is picked up and stored over an existing instruction at location B. When the program next reaches location B, control will be switched to location C. The transfer instruction at location A is dependent because its outcome is a function of its storing instruction. In this example the control flow problem of

Figure 3.5 - Dependent Instructions



a. The Transfer Switch b. The Subroutine Call c. The Calculated Transfer

d. The Changed Address e. The Indirect Address f. The Tagged Address determining which location receives control from the switch interacts with the data flow problems of detecting the switch and determining its location.

3.4.2 The Subroutine Call and Return

The second example of control flow - data flow interaction is the subroutine call and return. Figure 3.5b shows its general form. In this example, the subroutine is called by the calling instruction, TSX. The calling instruction is followed by a set of locations which form the subroutine calling sequence. The calling sequence set may be empty. The calling sequence is followed by a set of subroutine return locations, i.e. locations to which the subroutine transfers control when it is finished. Here too, the return set may be empty. The subroutine call and return sequence are dependent because its outcomes are a function of the subroutine itself. In this example the control flow problems of determining the length of the calling sequence and the number of return locations interact with the data flow problem of finding where and how the subroutine calculates its return.

3.4.3 The Calculated Transfer

The third example of a control flow - data flow interaction is the calculated transfer instruction. A calculated transfer occurs when a transfer instruction calculates its possible outcomes, i.e. the set of locations to which it transfers control. Figure 3.5c shows one of the forms of the calculated transfer, the tagged transfer. The tagged transfer uses its address and tag to determine which location receives control. Thus, the tagged transfer is a dependent instruction because its set of outcomes are a function of the index loading instruction. In this example the control flow problem of finding the set of locations which can receive control from the tagged transfer interacts with the data flow problem of finding where and how the index register is loaded.

3.4.4 <u>The Modified Instruction</u>

The fourth example of a control flow - data flow interaction is the modified instruction. A modified instruction occurs when a program modifies or changes a portion of an existing instruction. Figure 3.5d shows one of the many forms of the modified instruction. In this example the address portion of the instruction at location B is changed by the previous instruction. The instruction at location B is dependent because its outcome is a function of its modifying instruction. In this example the data flow problem of determining the new address portion of location B interacts with the control flow problem of finding which locations change the address portion of location B.

3.4.5 The Indirect Address

The fifth example of control flow - data flow interaction is the indirect addressed instruction. Figure 3.5e shows one of the forms of the indirect addressing. In this example the instruction at location A uses the address portion of location B to determine which location it references. The indirect address instruction at location A is dependent because its outcomes are a function of the instruction which last changed the address portion of location B. In this example the data flow problem of determining the address portion of location B interacts with the control flow problem of finding where that address was last changed.

3.4.6 The Tagged Address

The last example of control flow - data flow interaction is the tagged address instruction. A tagged address occurs when an instruction uses an index register to calculate its effective address. Figure 3.5f shows an example of a tagged address instruction. In this example the instruction at location A uses index register one to calculate which location is picked up from the table at location B. The tagged address instruction is dependent because its outcome is a function of the index loading instruction. In this example the data flow problem of deciding which location is picked out of the table interacts with the control flow problem of determining where the index register was last loaded.

CHAPTER 4

THE ANALYSIS SOLUTION

In the previous chapter a comparison of the properties of non-modifying and self-modifying programs led to the definition of independent and dependent instructions. The dependent instructions of self-modifying programs caused control flow - data flow interaction requiring an iterative analysis procedure. The problems introduced by iteratively applying the straight-forward, two-step analysis procedure for non-modifying programs were discussed.

This chapter presents the approximation procedures used by the first iteration to bootstrap itself through the control flow - data flow interaction cycle discussed in Chapter 3. First, the solution philosophy required for a successful analysis is stated. Second, the general organization of the first iteration is outlined. This outline describes the data acquisition and data processing sequence and shows the use of intermediate data flow analysis results to improve control flow approximations and vice versa. Finally, a more detailed presentation describes how the control and data flow steps handle the dependent instructions listed in Chapter 3.

4.1 THE SOLUTION PHILOSOPHY

If an automatic program analysis system is to be successful, it should be able to analyze long, core-length programs, such as assemblers and compilers. When long programs are analyzed, the analysis system may generate intermediate data tables that are at least two or three times as long as the original input program. Because it may not be possible to retain all of the intermediate tables in core, these results should be placed on external lists. Because of these large, external data lists, the analysis procedure should wherever possible consist of sorting, merging, and scanning. Any searching of these lists or other data structures should be avoided or delayed whenever possible. If this data processing philosophy is to be successful, a set of temporary result lists and a processing sequence must be developed.

4.2 THE FIRST ITERATION

Because the first iteration uses intermediate data flow analysis results to improve its control flow approximations and vice versa, a general outline of the first iteration organization would be helpful before the detailed dependent instruction solutions are discussed. The first iteration is divided into four parts: Data Gathering, Data Processing, Data Reduction, and Function Generation and Output. The organization and information processing are also graphically displayed in Figure 4.1 and Figure 4.2.

Figure 4.1 - The First Iteration Organization

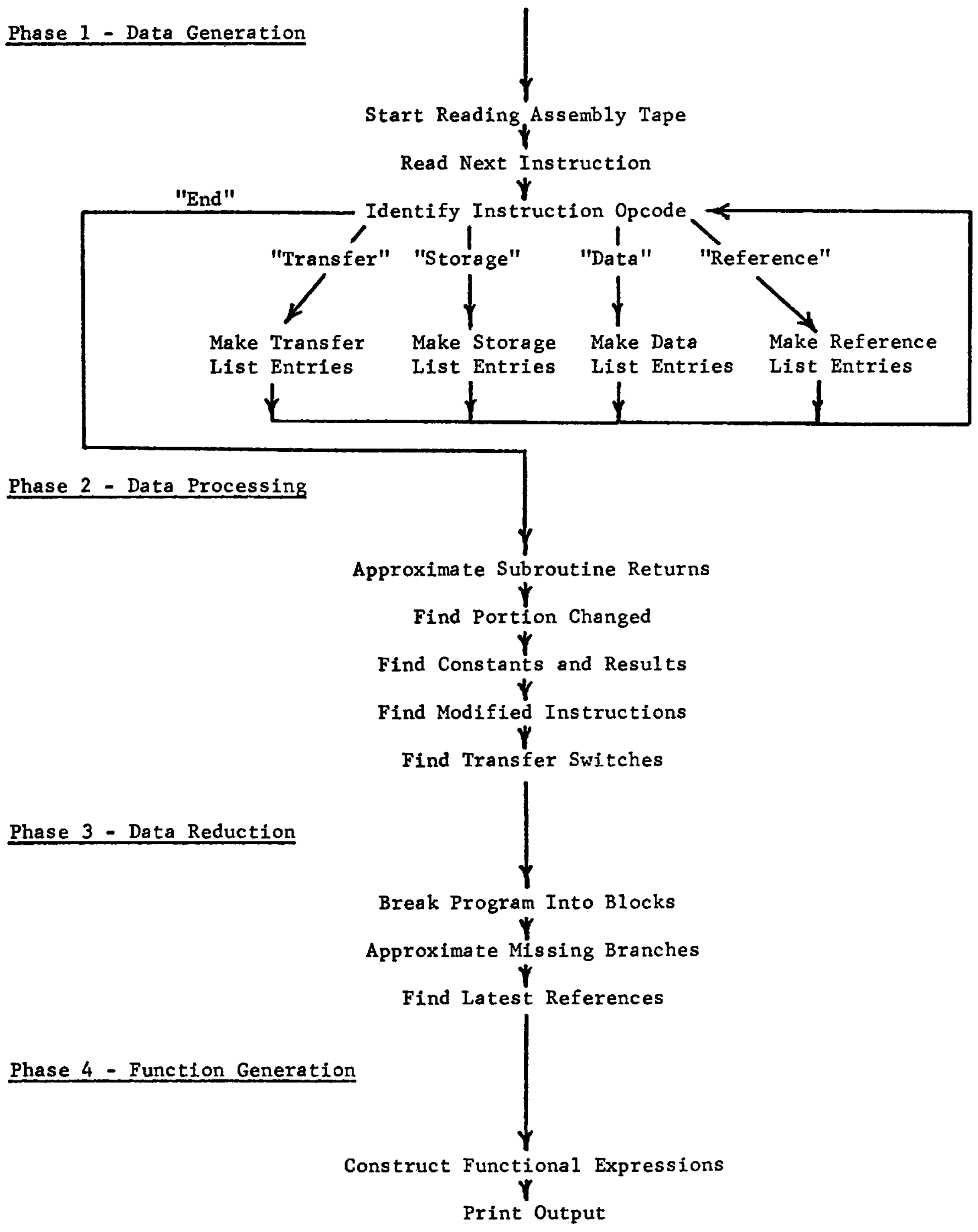
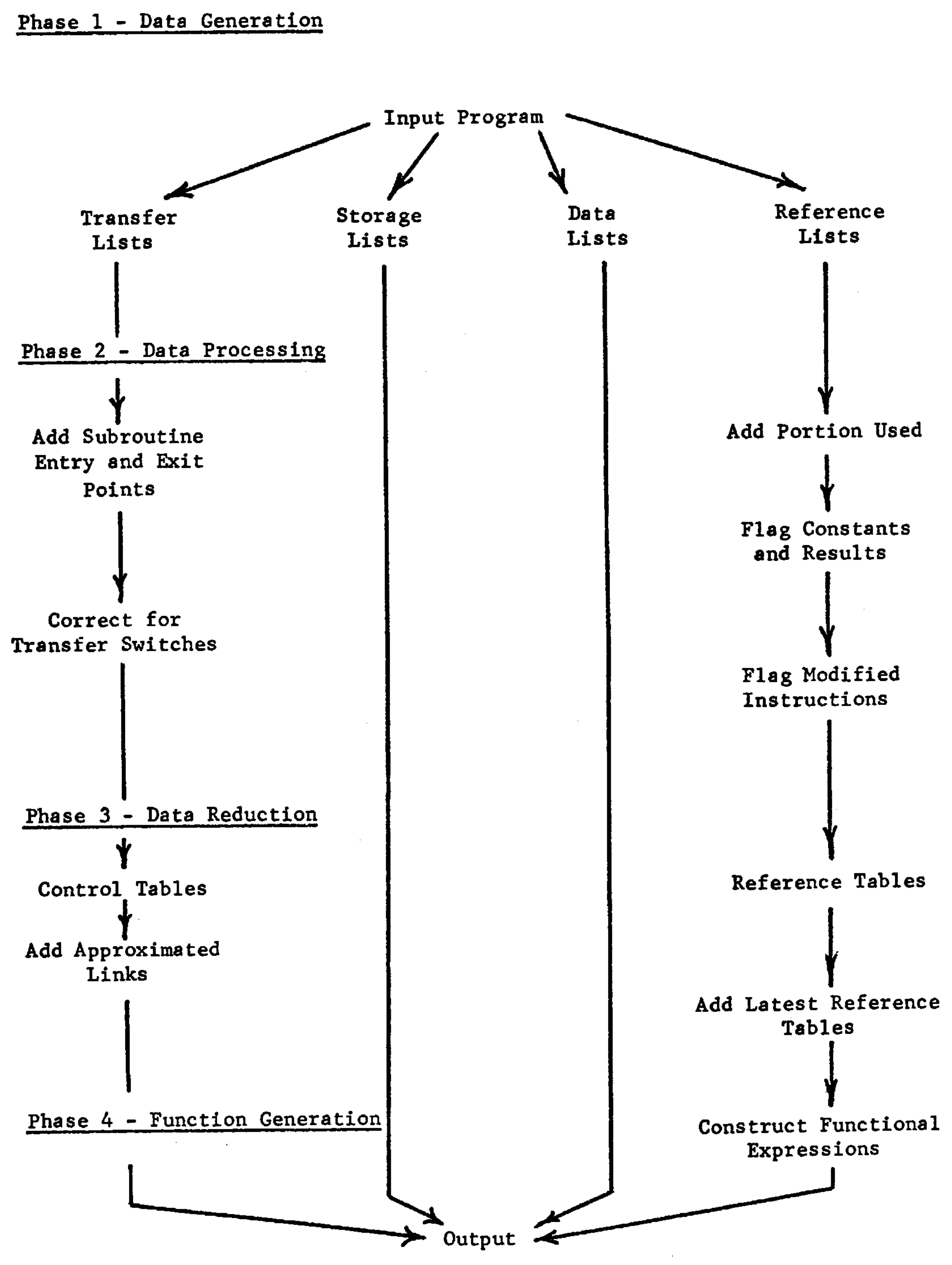

Phase 1 - Data Generation
Start Reading Assembly Tape
Read Next Instruction
"End"
Identify Instruction Opcode
"Transfer"
"Storage"
"Data"
"Reference"
Make Transfer List Entries
Make Storage List Entries
Make Data List Entries
Make Reference List Entries
Phase 2 - Data Processing
Approximate Subroutine Returns
Find Portion Changed
Find Constants and Results
Find Modified Instructions
Find Transfer Switches
Phase 3 - Data Reduction
Break Program Into Blocks
Approximate Missing Branches
Find Latest References
Phase 4 - Function Generation
Construct Functional Expressions
Print Output

Figure 4.2 - The First Iteration Information Processing


Phase 1 - Data Generation
Input Program
Transfer Lists
Storage Lists
Data Lists
Reference Lists
Phase 2 - Data Processing
Add Subroutine Entry and Exit Points
Correct for Transfer Switches
Add Portion Used
Flag Constants and Results
Flag Modified Instructions
Phase 3 - Data Reduction
Control Tables
Add Approximated Links
Reference Tables
Add Latest Reference Tables
Phase 4 - Function Generation
Construct Functional Expressions
Output

4.2.1 Data Gathering

The first phase transforms the input program from a set of assembly-language instructions into a set of temporary data lists. The input program is scanned one line at a time. First, the line is decoded and interrogated for such information as octal instruction, its assigned memory location, BCD instruction operation code, and absence or presence of a tag or indirect address. The assigned memory location and octal instruction were produced by the FAP assembler. They are used by the analysis program as bookkeeping aids for generating list or table entries, e.g. the assigned memory location is used in each table entry so that later analysis phases can determine which instruction originally generated the entry. The BCD operation code is used to decode the instruction because it permits some "interpretation" of programmer intent, e.g. data and storage pseudo-operations can be distinguished from executable instructions. Tagged and indirectly addressed instructions are detected so that special analysis procedures can be initiated.

Second, entries are added to the various data lists according to the BCD operation code. For transfer instructions, entries are added to the various Transfer Lists, e.g. the Entry and Exit Point Lists. For referencing instructions, entries are added to the Active and Passive Reference Lists. For data generation pseudo-operations, entries are added to the Data List. For storage generation pseudo-operations, entries are added to the Storage List, etc. Each list entry uses information decoded from the original instruction, e.g. if the instruction is tagged or indirectly addressed, special flags are set in its entries so as to alert later analysis phases.

4.2.2 Data Processing

The second phase determines program properties by using data processing techniques on the temporary data lists. In general, the lists are sorted to place them in proper order and then sequentially scanned to detect program properties.

First, general program properties are detected. Transfer Lists are sorted and scanned to determine first approximations to subroutine return points. These new entry and exit points are added to the Entry and Exit Point Lists. The Reference Lists are sorted and scanned to detect which portions of each cell are actively referenced; which cells are only passively referenced, i.e. constants; and which cells are only actively referenced, i.e. results.

Second, special program properties are determined. Modified instructions are detected by comparing each Active Reference List entry with those on the Data and Storage Lists. If a proper match is not found, the actively referenced location is flagged as a possible modified instruction. Possible transfer switch locations are found by comparing each entry on the Passive Reference List against all entries on the Exit Point List. A match indicates a passive reference to a location which contains a known transfer instruction. The matching Exit Point entry and the Reference Lists are then used to find a first approximation to the outcomes of the transfer switch. The new outcomes are added to the Entry and Exit Point Lists.

4.2.3 Data Reduction

The third phase transforms the processed temporary data lists into more convenient data structures. Generally, this involves sorting the lists into proper order and then placing each list entry into a new data structure by either scanning or searching the list.

First, the Transfer Lists which contain sorted entry and exit point information are transformed into Control Tables which represent the approximated control flow graph. The Entry and Exit Point Lists are used to break the program into blocks and to interconnect those blocks. This topological information is then represented in the Control Tables. Finally, the Control Tables are interrogated to detect unreachable blocks and to approximate and to insert missing control branches.

Second, the Reference Lists are resorted and transformed into Reference Tables by associating each Active and Passive Reference List entry with the block in which it occurs. Next, the "latest reference set" for each passive reference is found by searching the Control and Reference Tables. Finally, the latest reference information is placed into a suitable data structure.

4.2.4 Function Generation and Output

The fourth phase transforms the Latest Reference Tables into functional expressions and places those expressions in a suitable data structure for final output.

4.3 THE CONTROL FLOW SOLUTIONS

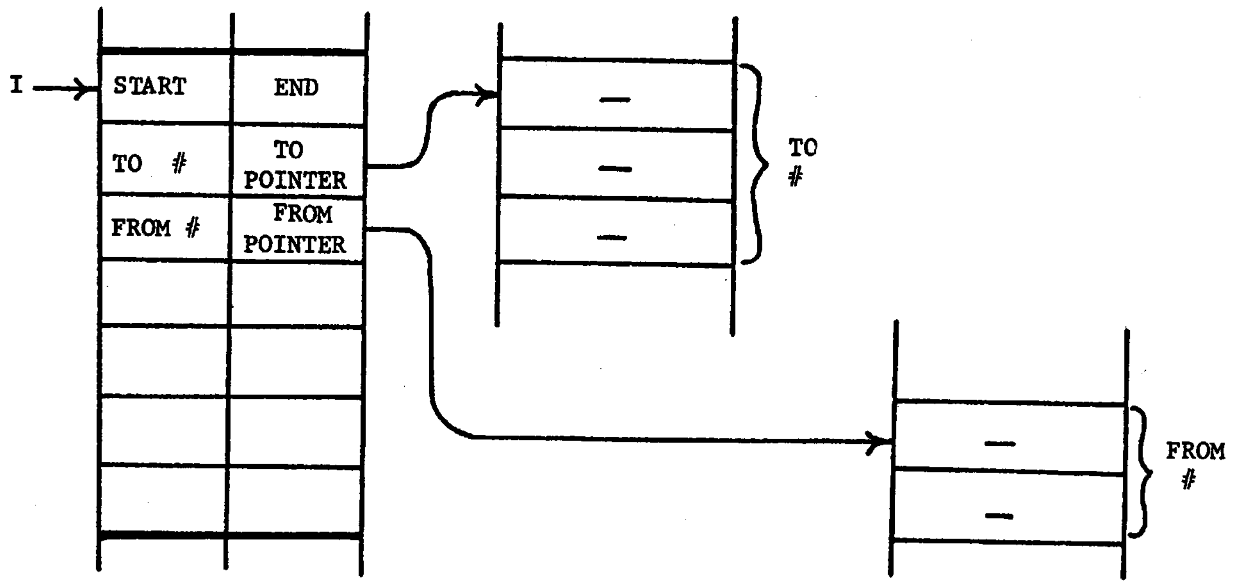
This section presents the solution techniques used to solve the control flow problems discussed in Chapter 3. First, the control flow graph structure is presented so that the end result is known in advance. This discussion includes the desired structure properties and a structure which incorporates those properties. Second, the solution techniques used to bootstrap through the dependent instruction interaction cycle are presented. These techniques include detecting the entry and exit points, determining the program blocks, and interconnecting the blocks.

4.3.1 The Control Graph Data Structure

The data structure which contains the control flow information must have two characteristics. First, the structure must permit forward and backward movement in the control flow graph. Forward, because the program is executed in that direction; backward, because the latest reference search is easier to program for that direction. Second, the structure must permit expansion and contraction of the control flow graph. Expansion, because later analysis iterations may detect new blocks; contraction, because those same iterations may wish to rejoin blocks.

A modification of Ross's plex (14) produces a data structure which incorporates the proper characteristics. The complete structure will be referred to as the Control Tables and is composed of three separate tables: the Topology Table, the To Table, and the From Table. Figure 4.3 shows the general component of each of these three tables.

Figure 4.3 - The Control Tables



a. Topology Table b. To Table c. From Table

The Topology Table serves as the "card catalogue" for analysis results. When the analysis program needs information about a given block, it can be found through the Topology Table once the Block Number, I, is known. The Topology Table entries are numbered sequentially with the starting program block coming first, the second block second, etc. A Topology Table entry is composed of seven sequential words. The first word contains the STARTing and ENDing location of the particular block. The second word is the "catalogue card" for the blocks which can be reached from the particular block. The left half contains the count of those blocks, and the right half points into the To Table where the Block Numbers of those reachable blocks are stored. The third word is the "catalogue card" for the blocks which can pass control to this particular block and is constructed similarly to the second word. The fourth through seventh words are reserved for data flow information and will be discussed in a later section.

The To Table contains a variable length entry containing the Block Number of each block reachable from the given block. Likewise, the From Table contains a variable length entry containing the Block Number of each block which can pass control to the given block.

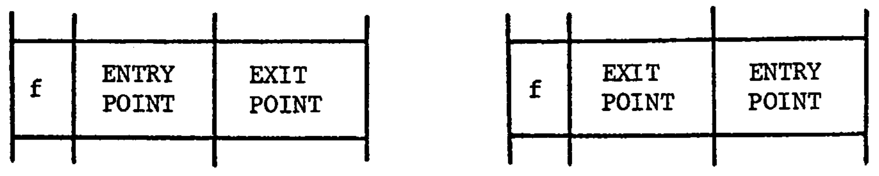
4.3.2 Detecting the Entry and Exit Points

During the Data Gathering Phase, entries are added to the temporary Transfer Lists whenever a transfer or control type instruction is found.

If the data structure of these lists is to conform with the general solution philosophy discussed earlier, the structure must permit individual entries to be added as required but yet allow all entries to be processed as a group.

These characteristics can be incorporated into two lists, the Entry Point List and the Exit Point List. The Entry Point List contains the entry point entries, and the Exit Point List contains the exit point entries. The format of the list entries is shown in Figure 4.4. The "f" portion of each entry retains information about the function or purpose of the transfer instruction which generated the entry, e.g. remembers that the instruction was an absolute transfer, a subroutine call, or a tagged transfer. The "Entry Point" portion of each entry contains the core location of the entry point. The "Exit Point" portion of each entry contains the core location of the exit point.

Figure 4.4 - The Entry and Exit Point List Formats

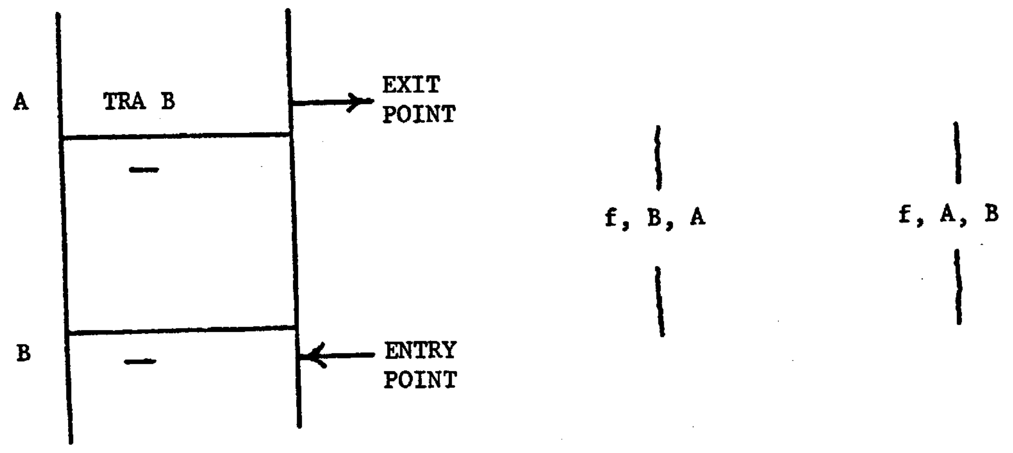


a. Entry Point List

b. Exit Point List

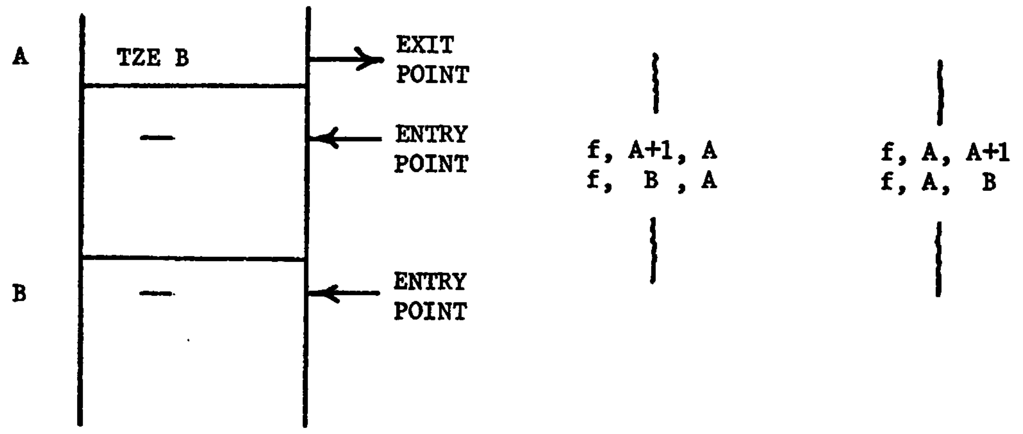
Generating the Entry and Exit Point List entries involves detecting all control instructions and determining their outcome sets. The outcome of an independent control instruction can be determined from the instruction itself. Figures 4.5 and 4.6 show examples of list entries generated by independent instructions during the Data Gathering Phase. Note that, except in special cases which are discussed later, Entry and Exit Point List entries are made in pairs. This procedure facilitates breaking the program into blocks. However, there is a small but important percentage of control instructions which are dependent and whose outcome sets cannot be determined by the Data Gathering Phase. Now, three such dependent instructions are discussed to indicate how their Entry and Exit Point List entries are generated.

Figure 4.5 - The Entry and Exit Point Entries of an Absolute Transfer



a. The Program b. The Entry List c. The Exit List

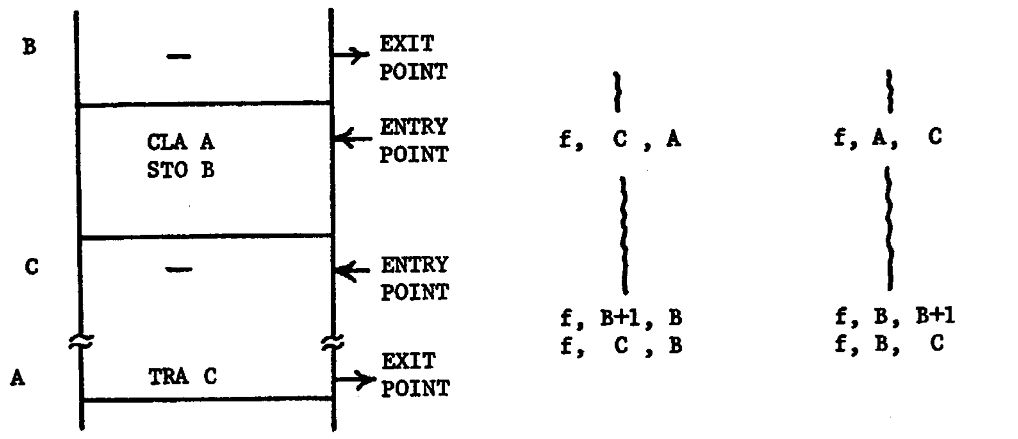
Figure 4.6 - The Entry and Exit Point Entries of a Conditional Transfer



a. The Program b. The Entry List c. The Exit List

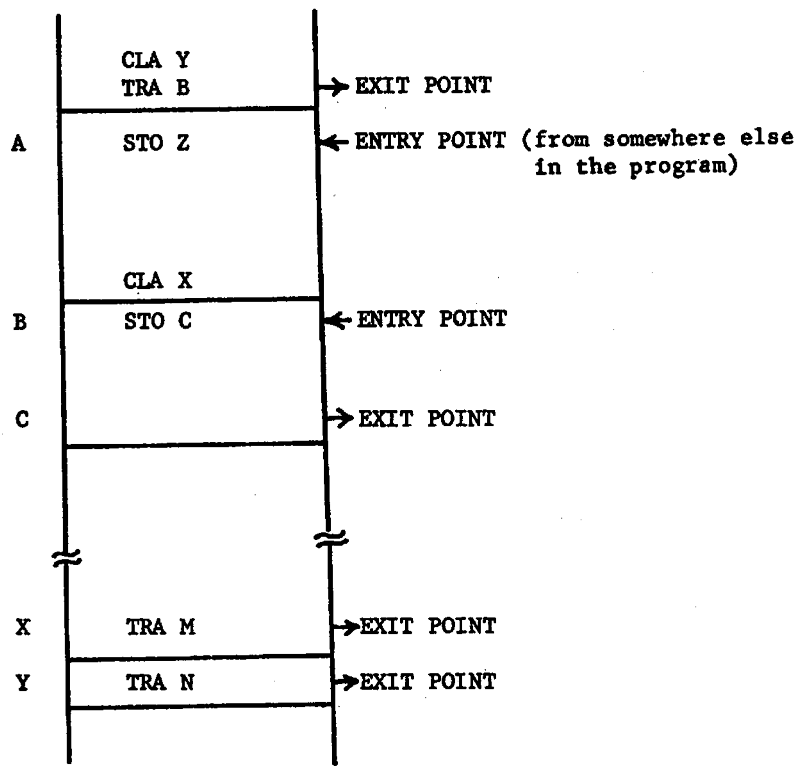
The first example of a dependent control instruction is the Transfer Switch. Figure 4.7a shows how a Transfer Switch might occur in a program. During the Data Generation Phase, Entry and Exit Point List entries are made for the TRA C instruction, and Active and Passive Reference List entries are made for the CLA A and STO B instructions. During the Data Processing Phase, the analysis program detects a passive reference to a location containing a transfer instruction. In this case the Passive Reference List contains a passive reference to location A generated by the CLA A instruction, and the Exit Point List contains an entry at location A generated by the TRA C instruction. Thus, the Data Processing Phase knows that the CLA A instruction fills the accumulator with an

Figure 4.7 - The Transfer Switch



a. The Program b. The Entry List c. The Exit List

Figure 4.8 - Transfer Switch with Passive-Active Reference Separation

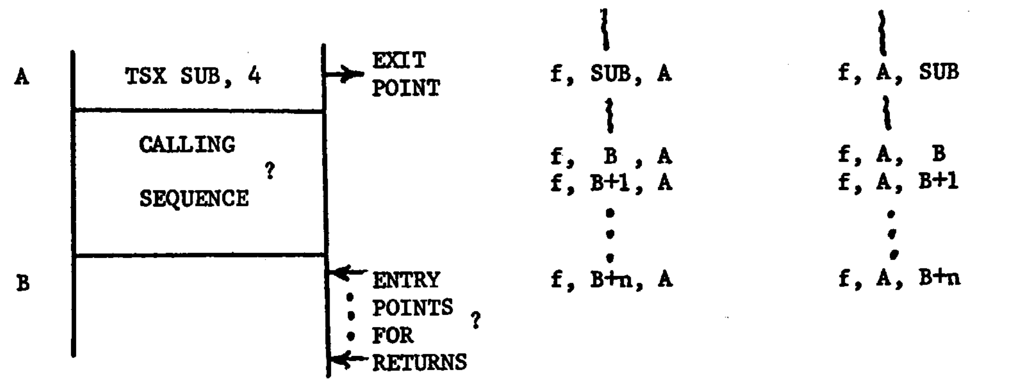


instruction that passes control to location C. It also knows that the instruction is at location A and the "f" portion of its Exit Point List entry indicates an absolute transfer, TRA. The Data Processing Phase determines where the accumulator stores the transfer instruction by noting that the "next" passive reference to the AC after the active reference to the AC generated by the CLA A instruction is the STO B instruction. Therefore, since the STO B instruction actively references location B, the transfer instruction is stored into B. Because control can be split two ways at location B, two entry point - exit point pairs are added to the end of the lists as shown in Figures 4.7b and 4.7c. The "f" portions of these new entries indicate generation by a Transfer Switch. Note that care must be taken to determine whether or not the passive reference which picks up the transfer is separated from the active reference which stores the transfer by either an entry or exit point. If the references are separated, the "correct" active reference cannot be found until after the first approximation to the control flow has been determined, i.e. during the second iteration. Figure 4.8 shows such a case. The TRA N instruction is stored into location C, not Z. Finally, the Data Processing Phase must determine whether the transfer instruction which causes the switch can be executed in its original location. This is done by seeing if there is a data or storage pseudo-operation on the Data or Storage Lists in a location "just above" the location of the transfer instruction. If there is, the Entry and Exit Point List entries originally generated by the transfer are removed because the transfer instruction "appears" to be included in a "data area" and is "probably" not executed in its original location.

The second example of a dependent control instruction is the Subroutine Call and Return. Figure 4.9a shows how a subroutine call can occur in a program. Subroutine return points must be found so that the proper Entry and Exit Point List entries are made and the program can later be broken into the correct blocks. For analysis purposes, there are two types of subroutines. The first type is the external subroutine which is assembled separately from its calling program and need not be available for analysis. An external subroutine can be detected by a call which transfers control to a location in the Transfer Vector, i.e. a location before the first executable instruction. The external subroutine return information must be supplied as input information along with the original input program. This information is processed during the Data Gathering Phase and is used to generate Entry and Exit Point List entries.

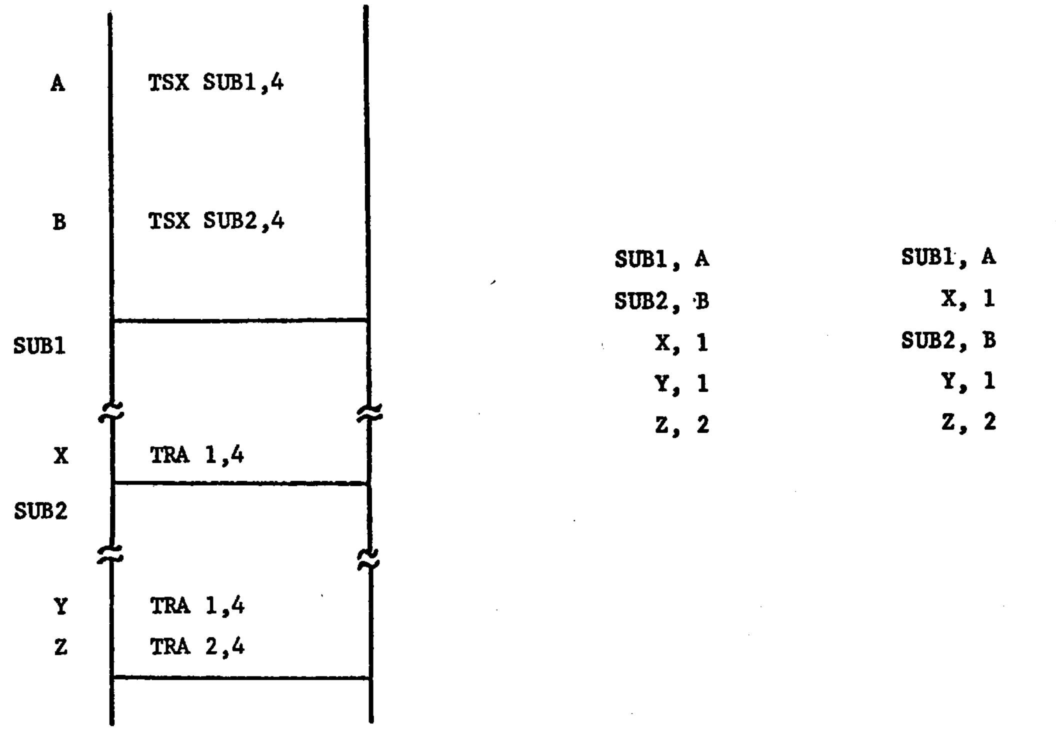
The second type of subroutine is the internal subroutine. It is assembled along with its calling program and is available for analysis. During the Data Gathering Phase, a Subroutine Return List containing internal subroutine calls and probable subroutine returns is constructed. A subroutine is <u>usually</u> called in the FAP language by a TSX instruction. A subroutine <u>usually</u> returns via a tagged, absolute transfer, such as a TRA "small constant", 4. When a TSX instruction is found, a call entry is added to the end of the Return List; when a probable subroutine return

Figure 4.9 - The Subroutine Call and Return



a. The Program b. The Entry List c. The Exit List

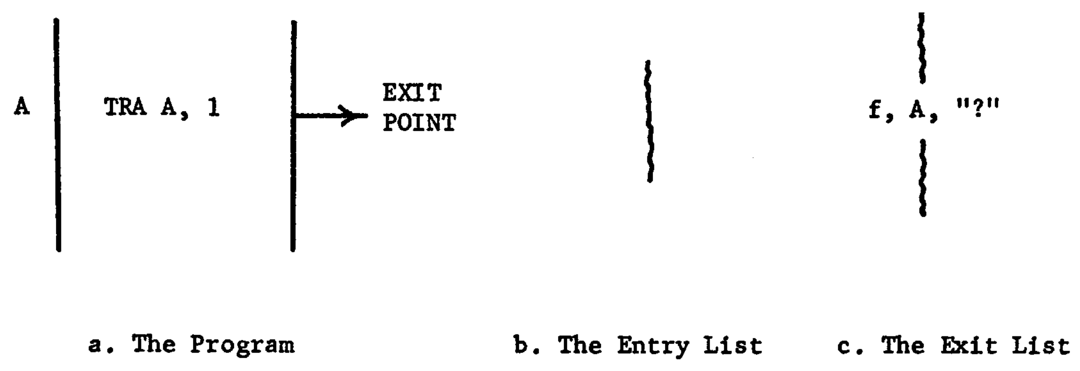
Figure 4.10 - The Subroutine Return List



a. The Program b. The Return List c. The Sorted Return List instruction is found, a return entry is added to the end of the Return List. Figure 4.10a shows an example of a program; Figure 4.10b shows its Subroutine Return List; and Figure 4.10c shows its sorted Return List. Note that in the sorted list, the returns for each subroutine are grouped together under its entry point or starting location. This technique assumes that all instructions of each subroutine are sequentially grouped together, e.g. SUB1 and SUB2 do not have any common instructions in Figure 4.10a. If subroutines do have common instructions, this approximation procedure produces invalid return points which must be corrected after the first approximation to control flow has been determined, i.e. in a later iteration. Figures 4.9b and 4.9c show how the entry point and exit point entries are added to the end of the lists for each subroutine call.

The third example of a dependent instruction is the calculated transfer. Figure 4.11 shows how one form of the calculated transfer, the tagged transfer, might occur in a program. Note that the tagged transfer in Figure 4.11 has a symbolic or relocatable address and is "probably" not a subroutine return. During the Data Generation Phase, only the location of the Exit Point is known, i.e. the location of the tagged transfer instruction. Therefore, only a single Exit Point List entry can be made and is shown in Figure 4.11c. Its "f" portion shows a tagged transfer, and its "Entry Point" portion is flagged as unknown. The problem of the missing entry points is passed on to later analysis phases.

Figure 4.11 - The Calculated Transfer



4.3.3 Determining the Program Blocks

After the Data Generation and Data Processing Phases detect the control instructions and generate the Transfer List entries, the Data Reduction Phase uses the lists to determine the program blocks. First, the lists must be ordered. The Entry Point List is sorted on its "Entry Point" column; the Exit Point List is sorted on its "Exit Point" column. Second, the program is broken into blocks by sequentially scanning the two lists and recognizing the various entry and exit point patterns.

There are four different types of blocks which produce four different Entry and Exit Point patterns. These are:

1. Blocks with both entry and exit points,
2. Blocks with only exit points,
3. Blocks with only entry points, and
4. Blocks with neither entry points nor exit points.

The patterns are recognized by detecting the occurrence of certain mathematical relationships between the "Entry Point" portion of the sorted Entry Point List entries and the "Exit Point" portion of the sorted Exit Point List entries. Each list has its own pointer which specifies the current entry on the list, e.g. the Entry Point List Pointer specifies the Current Entry Point. The term, Next Entry Point, refers to the next different entry after the current entry. Since both lists have been sorted, it is always true that the Next Entry Point be greater than the Current Entry Point. Likewise, the next Exit Point must be greater than the Current Exit Point. As the respective entries are processed, the pointers are moved down the lists. The recognition process is recursive, and the recognition expressions stated below assume that all entries and exits for the previous block have been processed.

1. The current block has both entries and exits:

   Current Entry = Previous Exit + "1"

   Current Entry < Current Exit

   Current Exit < Next entry

2. The current block has only exits:

   Current Entry ≠ Previous Exit + "1"

   Current Entry > Current Exit

3. The current block has only entries:

   Current Entry = Previous Exit + "1"

   Current Exit > Next Entry

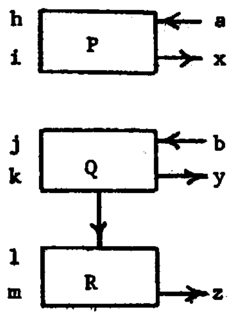
4. The current block has neither entries nor exits:

   Current Entry ≠ Previous Exit + "1"

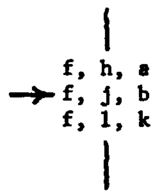
   Current Entry < Current Exit

Figure 4.12a shows a flowchart outline which contains a block with both entries and exits. Block Q can be reached from location b and transfers control to locations l and y. Block Q starts at location j and ends at location k. Figures 4.12b and 4.12c shows the Sorted Entry and Exit Lists. If Block P has already been formed, then the arrows on the two sorted lists point to the current list entries. Block Q has both entries and exits because the list entries satisfy the first set of relationships shown above, i.e. $j = i + 1$, $j < k$, and $k < l$. The START of Block Q is j, and the END is k. Figures 4.12c, d, and e show

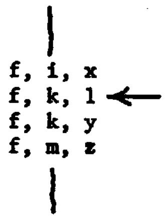
Figure 4.12 - A Block with both Entry Point and Exit Point Entries



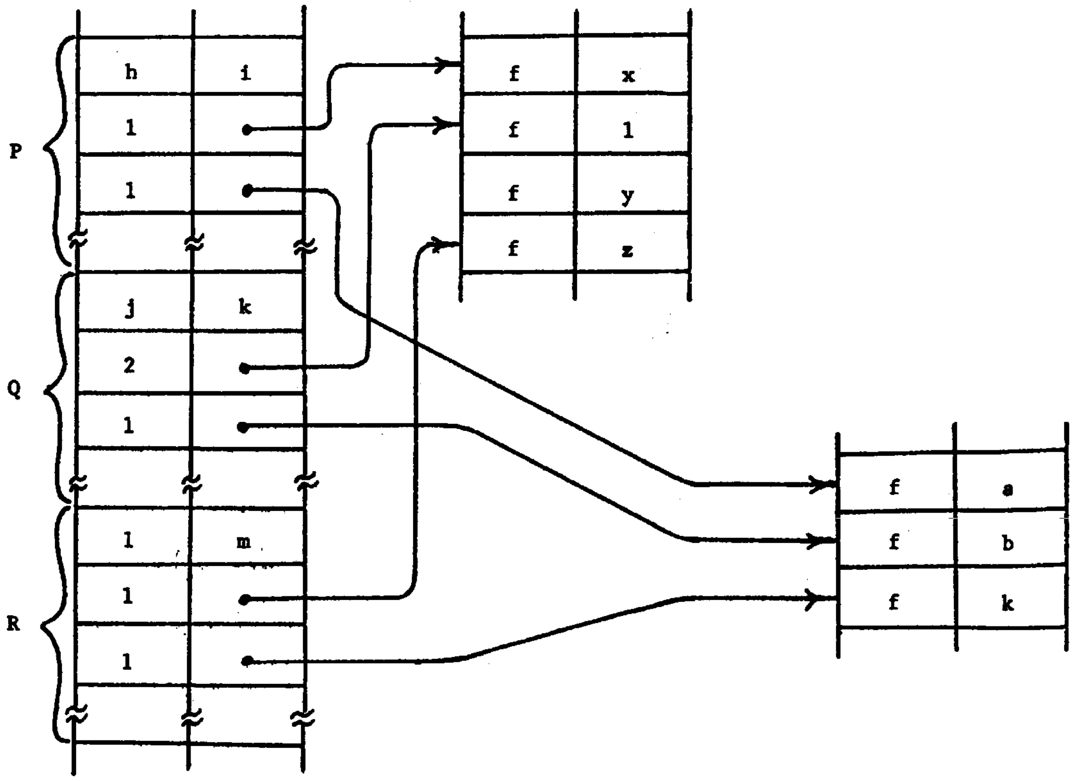
a. The Program



b. Sorted Entry List



c. Sorted Exit List



d. The Topology Table

e. The To Table

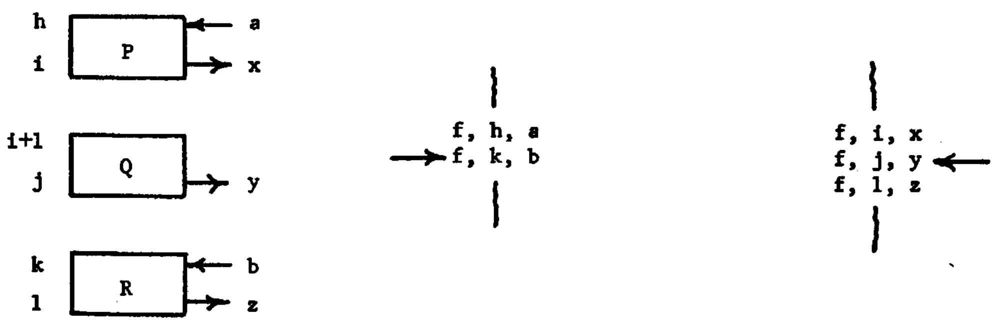
f. The From Table

the Control Tables for Block Q. Since there are two Exit List entries with an "Exit Point" portion of k, there are two To Table entries, l and y. Since there is only one Entry List entry with an "Entry Point" portion of j, there is only one From Table entry, b. In this example and those to follow, the entries in the To and From Tables are core locations, not Block Numbers. The core locations are replaced by Block Numbers after the program has been broken into blocks.

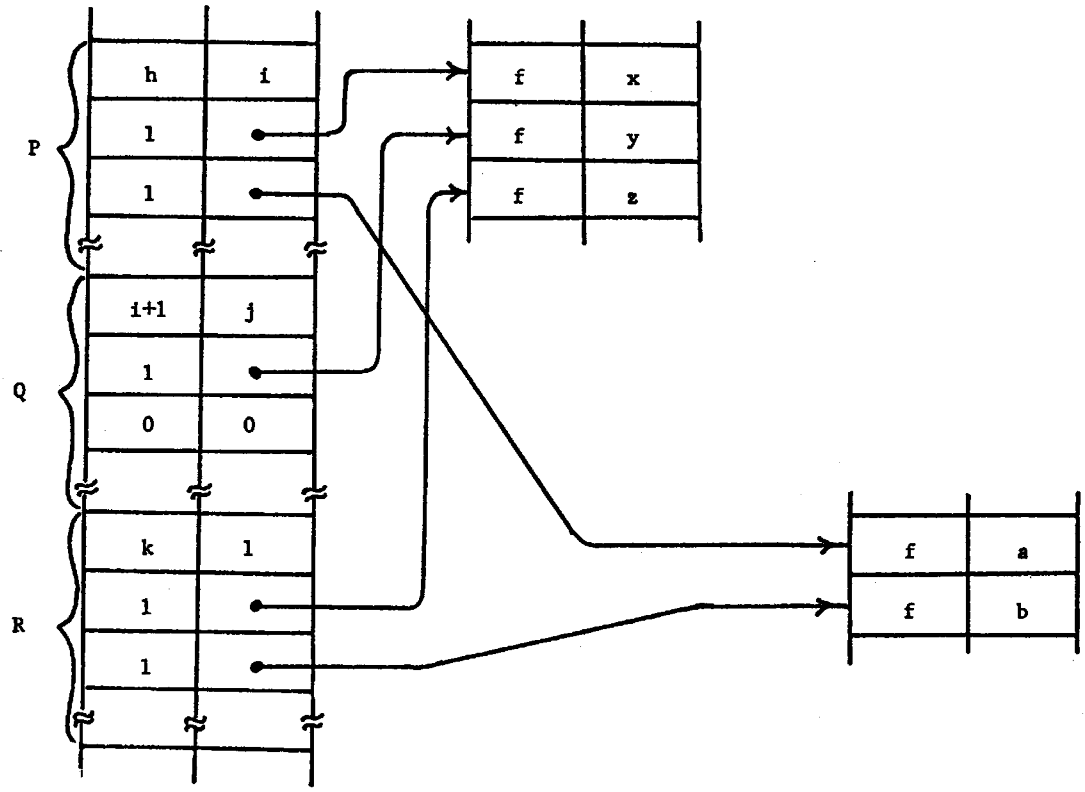
Figure 4.13a shows a flowchart outline which contains a block with only exits. Block Q only exits to location y. (The entry at $i + 1$ can be missing because of a calculated transfer not generating its entry point entries during the Data Generation Phase.) Block Q starts at location $i + 1$ and ends at location j. Figures 4.13b and 4.13c show the Sorted Entry and Exit Lists. If Block P has already been formed, then the arrows on the two sorted lists point to the current list entries. Block Q has only exits because the list entries satisfy the second set of relationships shown above, i.e. $k \neq i + 1$ and $k > j$. The START of Block Q is $i + 1$, and the END is j. Figures 4.13c, d, and e show the Control Tables for Block Q. Since there is one Exit List entry with an "Exit Point" portion of j, there is one To Table entry, y. Since there are no Entry List entries with an "Entry Point" portion of $i + 1$, there are no From Table entries for Block Q.

Figure 4.14a shows a flowchart outline which contains a block with only Entry List entries. Block Q receives control from location i, but transfers control directly to the next sequential block. Figures 4.14b

Figure 4.13 - A Block with only Exit Point Entries



a. The Program b. Sorted Entry List c. Sorted Exit List



d. Topology Table e. To Table f. From Table and 4.14c show the Sorted Entry and Exit Lists. If Block P has already been formed, then the arrows on the two lists point to the current list entries. Block Q has only entries because the list entries satisfy the third set of relationships shown above, i.e. $j = i + 1$ and $1 > k$. The START of Block Q if j, and the END is $k - 1$. Figures 4.14c, d, and e show the Control Tables for Block Q. Since Block Q exits directly to the next block, an exit is inserted from location $k - 1$ to location k. Thus Block Q has one To Table entry, k. Note that Block R has two From Table entries, b and $k - 1$. Since there is one Entry List entry with an "Entry Portion" of j, there is one From Table entry, i.

Figure 4.15a shows a flowchart outline which contains a block with neither entry not exit points. Figures 4.15b and 4.15c show the Sorted Entry and Exit Lists. If Block P has already been formed, then the arrows on the two lists point to the current list entries. Block Q has neither entries nor exits because the list entries satisfy the fourth set of relationships shown above, i.e. $j \neq i + 1$ and $j < k$. The START of Block Q is $i + 1$, the END is $j - 1$. There are no To or From Table entries.

4.3.4 Interconnecting the Blocks

In the previous section, techniques for breaking the program into blocks and constructing the Control Tables were described. Now, these tables must be checked to insure that the blocks have been properly

Figure 4.14 - A Block with only Entry Point Entries

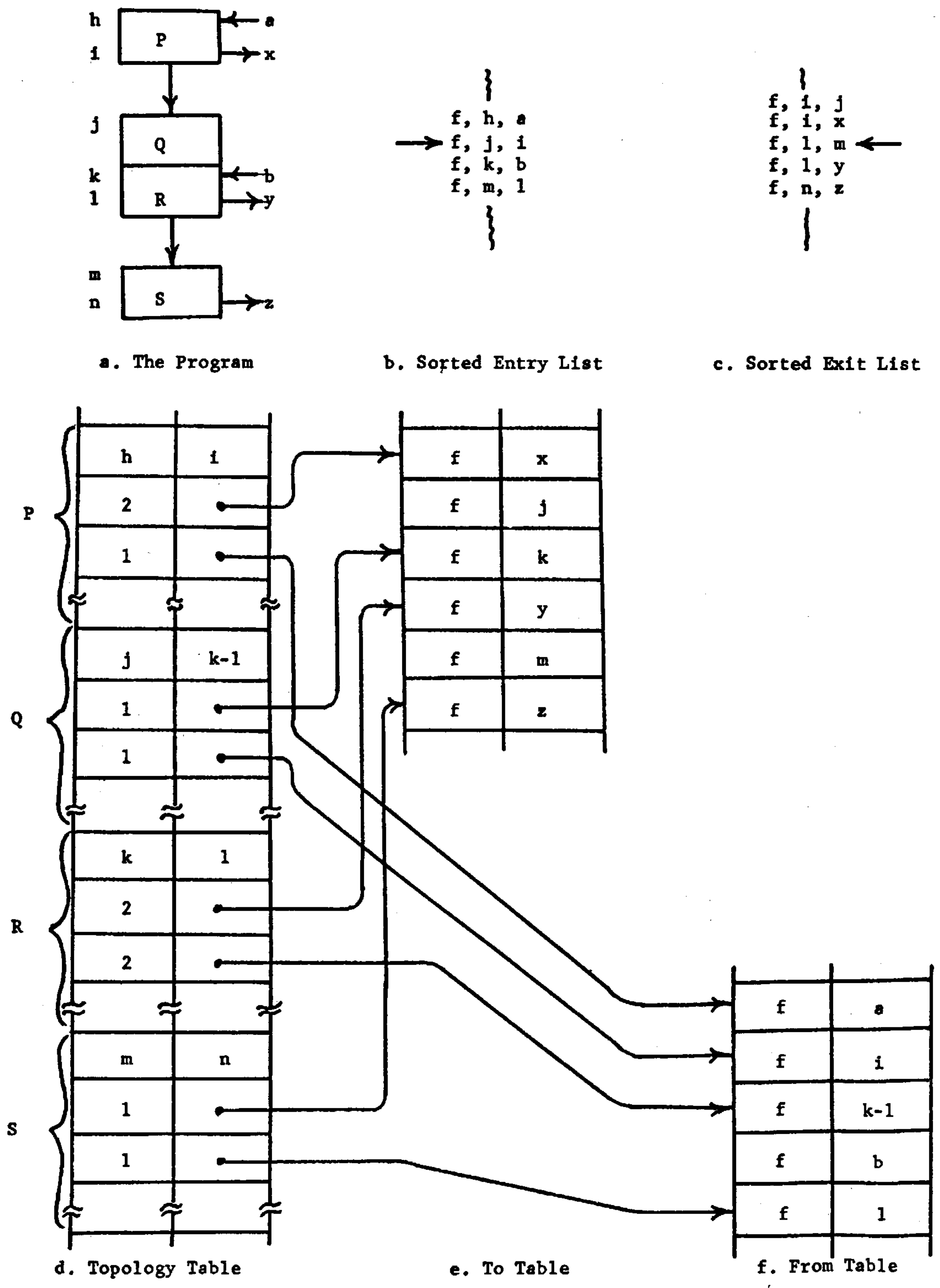
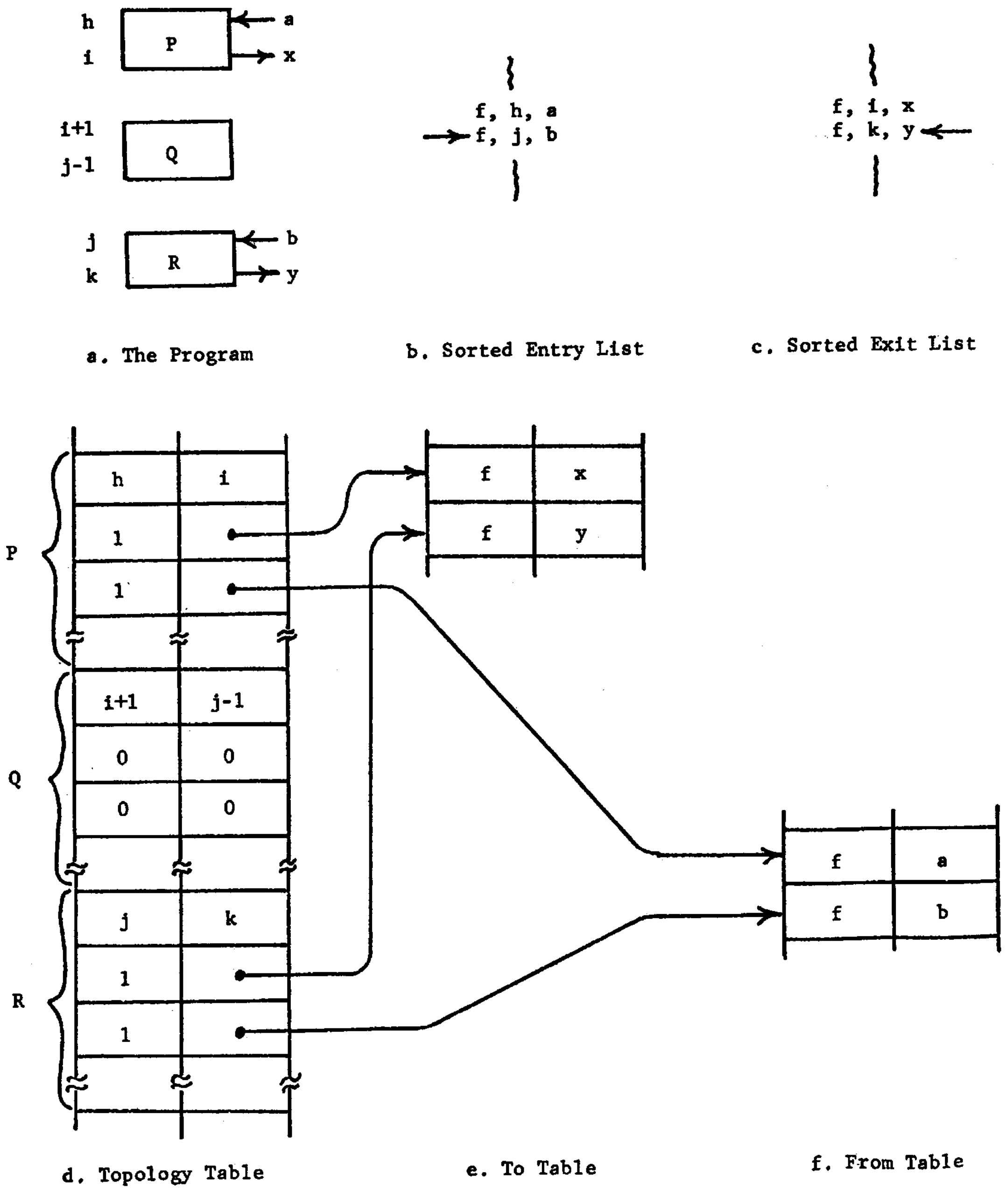


Figure 4.15 - A Block with neither Entry Point nor Exit Point Entries



and totally interconnected so that all program blocks are used in the analysis. The purpose of this section is to discuss techniques for testing block interconnections, detecting isolated or improperly connected blocks, and correcting improper block connections.

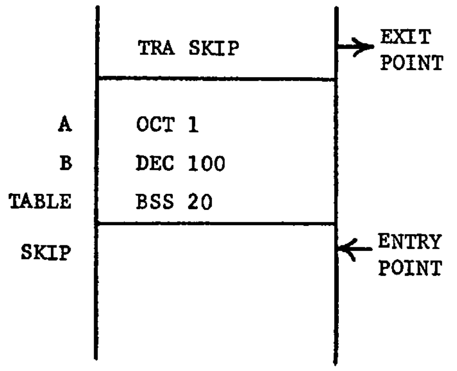
The program being analyzed must be assumed to be a "well connected" program where each program block can be reached from at least one of the program starting blocks. (A subroutine can have any number of starting blocks or entry points.) If a block cannot be reached from a starting block, there must be some reason for its isolation. Detecting isolated blocks first requires constructing a list of blocks which can be reached from one of the starting blocks and then determining which blocks are missing from this reachable block list.

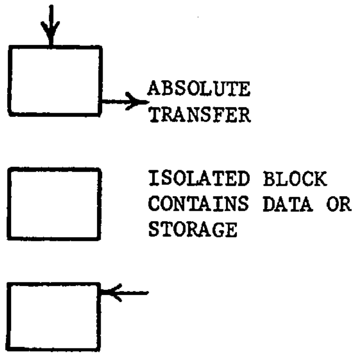
As each isolated block is detected, the reason for its isolation must be determined; and its Control Table entries corrected. If the block should be isolated, its Topology Table entry is flagged as such. However, if the block should not be isolated, the proper assumed connection branches must be inserted into the Control Tables to make the isolated block reachable from its true predecessor blocks. After the Control Tables have been corrected for the isolated block, a new list of reachable blocks is constructed; and the detection procedure is repeated. This detection and correction procedure is repeated until all blocks are either reachable or flagged as truly isolated. Because of the generality of assembly-language programming, there are many different reasons for isolated blocks. It is at this point that individual algorithms must be developed for each class of reasons.

Probably the most common reason why a block should be isolated is that it contains data or storage pseudo-operations and is not meant to be executed. (Of course, there will always be the programmer who, for reasons known only to himself, uses data or storage pseudo-operations to generate executable code.) Figure 4.16a shows the structure of a program containing such a block. If this type of block is found missing from the reachable block list, its reason for being isolated can be verified as follows. First, the Data and Storage Lists are scanned to see if they contain at least one entry whose program location places it within the isolated block. Second, the Control Table entry of the block preceding the isolated block is checked to see if it is terminated by a single absolute transfer. If both conditions are satisfied, the block is truly isolated; and a data or storage flag is set in its Topology Table entry.

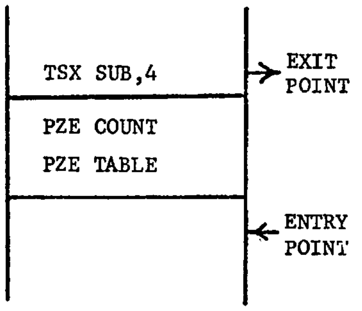
Another common reason why a block should be isolated is that it contains a subroutine calling sequence. Figure 4.16b shows the structure of a program containing such a block. If this type of block is found missing from the reachable block list, the To Table entry of the preceding block must show that it is terminated by a subroutine call. Because of the generality of assembly-language programming, a calling sequence can contain any type of instruction or pseudo-operation. At this stage of analysis, the isolated block can only be flagged as an assumed calling sequence. In a later iteration after the subroutine return approximations have been verified, the interaction between subroutine

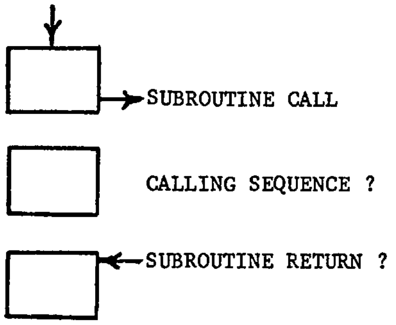
Figure 4.16 - Isolated Blocks



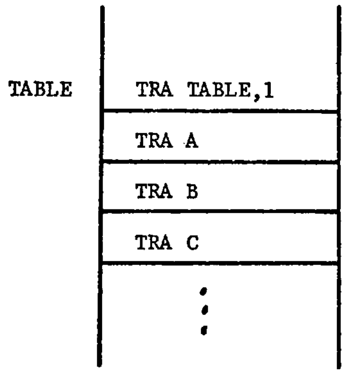


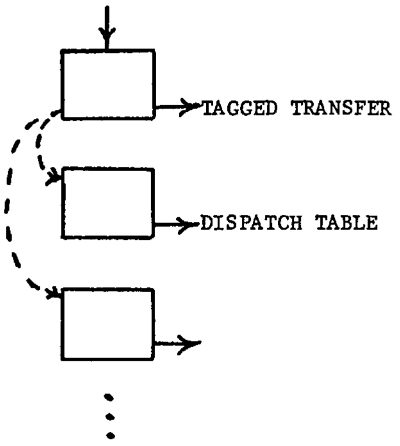
a. The Data or Storage Block





b. The Calling Sequence Block





c. The Dispatch Table

calling sequences and subroutine returns can be used to verify the flagging of blocks as calling sequences.

One common use of the calculated transfer is in a dispatch table. A dispatch table is a sequential set of blocks where the first block is terminated by a tagged transfer and the other blocks are terminated by an absolute transfer. The contents of the index register of the tagged transfer are used to determine which dispatch table block receives control from the tagged transfer block. Figure 4.16c shows a program containing a dispatch table. When the program is broken apart, the blocks in the dispatch table are formed as a function of exit points alone, because the entry points of the tagged transfer are missing. Thus, no connections are made between the tagged transfer block and the dispatch blocks. When a reachable block list is constructed, the dispatch blocks and those connected to them are missing. Therefore, assumed branches must be inserted into the Control Tables to interconnect the tagged transfer block and the dispatch blocks as shown by the dashed arrows in Figure 4.16c. These assumed branches permit the analysis program to reach the blocks which are connected to the dispatch blocks. In a later iteration after the set of possible index register values has been determined, the assumed branches can be verified.

4.4 THE DATA FLOW SOLUTIONS

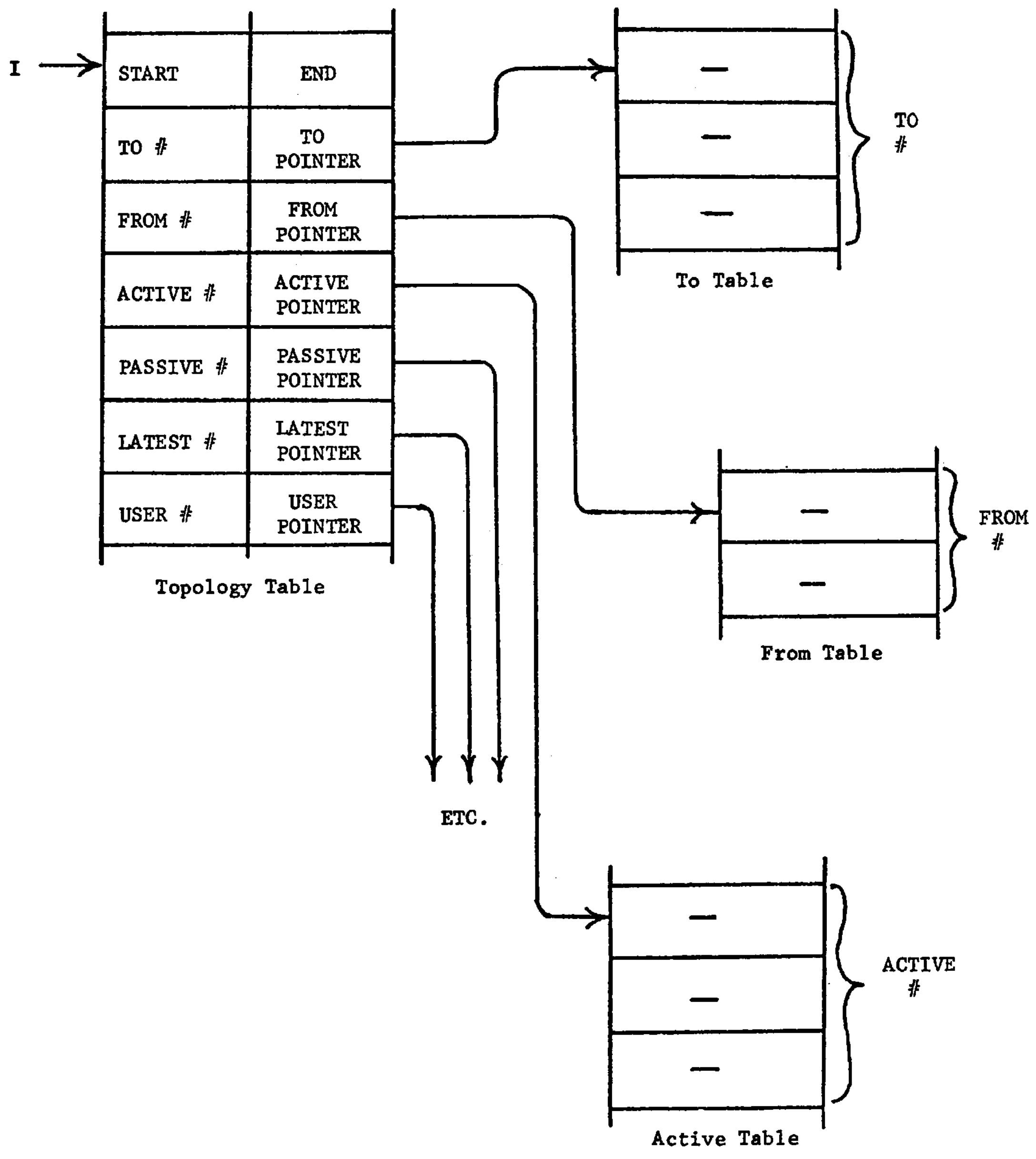
This section presents the solution techniques used to solve the data flow problems discussed in Chapter 3. First, the data flow graph data structure is presented so that the end result is known in advance. This discussion includes the desired structure properties and a structure which incorporates those properties. Second, the solution techniques used to bootstrap through the dependent instruction interaction cycle are presented. These techniques include generating the active and passive references, finding the latest reference sets, saving the latest reference information, and constructing the functional expressions.

4.4.1 The Data Graph Data Structure

The data structure which contains the data flow information must have three characteristics. First, the data flow information should be incorporated into the control flow structure so that the latest reference searches can be easily performed. Second, the structure should permit the Active and Passive Reference List entries to be associated with the block in which they occur in order to facilitate the latest reference searches. Third, the structure should retain the latest reference information in such a way as to provide for passing the functional expressions generated by each active reference on to those passive references which will need the expressions.

The first two desired characteristics can be accomplished by enlarging the Topology Table entry for each block to include "catalogue cards" for data flow information. Figure 4.17 shows the enlarged Topology Table block entry. The construction and interpretation of the

Figure 4.17 - The Enlarged Topology Table



new table words are the same as before, i.e. the left side gives the table entry count, and the right side points to those entries in the given table. The third characteristic can be fulfilled by properly constructing the four new tables, i.e. the Active, Passive, Latest, and User Tables. The format and construction of these tables will be introduced as they are needed.

4.4.2 Generating the Active and Passive References

The purpose of the Reference List entries is to tell the later analysis phases what and where information is changed, used or needed. The Data Generation and the Data Processing Phases construct the individual active and passive reference entries. The Data Reduction Phase uses the active references to find the latest reference sets for each passive reference. The Functional Generation Phase uses the latest references to construct the functional expressions for inside the flowchart boxes. If this processing chain is to be successful, the initial Reference Lists must be properly constructed.

During the Data Gathering Phase, entries are added to the temporary Reference Lists whenever an instruction which changes or uses information is found. If the data structure of these lists is to conform with the general solution philosophy discussed earlier, the structure must permit individual entries to be added as required but yet allow all entries to be processed as a group.

These characteristics can be incorporated into two lists, the Active Reference List and the Passive Reference List. The Active Reference List contains the active reference entries, and the Passive Reference List contains the passive reference entries. The format of the list entries is shown in Figure 4.18. The "f" portion of each entry retains information about the function or purpose of the instruction which generated the reference entry, e.g. remembers that the instruction was a plain STA instruction which changes only the address portion of the "Cell Changed"; was a CLA instruction with a symbolic operand address of **; or was a tagged STO instruction. The "Cell Changed" portion of an active reference entry is the cell number of the cell changed by the active reference. (For bookkeeping purposes, the central processor registers are also assigned cell numbers.) The "Cell Used" portion of a passive reference entry is the cell number of the cell used by the passive reference. The "Instruction Cell" portion of both entry types is the cell number of the cell which contains the instruction which generated the entries.

Generating the Active and Passive Reference List entries involves detecting all referencing instructions and determining their outcome sets. The outcome of an independent reference instruction can be determined from the instruction itself. Figures 4.19 and 4.20 show examples of list entries generated by independent instructions during the Data Gathering Phase. The number of entries made for each instruction is a function of its operation code. As in the case of control

Figure 4.18 - The Active and Passive Reference List Formats

f	CELL CHANGED	INSTRUCTION CELL

a. The Active Reference List

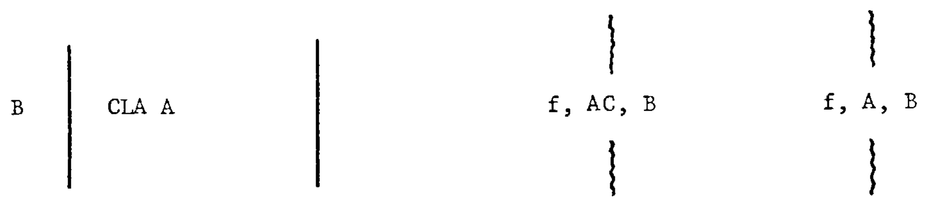
f	CELL USED	INSTRUCTION CELL

b. The Passive Reference List

instructions, there is a small but important percentage of referencing instructions which are dependent and whose outcome sets cannot be determined by the Data Gathering Phase. Now, three such dependent instructions are discussed to indicate how their Active and Passive Reference List entries are generated. This discussion shows how special Reference List entries are used to initiate special procedures to handle dependent instructions.

The first example of a dependent reference instruction is the changed address instruction. Figure 4.21a shows how a changed address instruction might occur in a program. Since the Data Generation Phase has no way of knowing in advance that the instruction at location B is modified, the Data Generation Phase generates the normal Reference List entries for that instruction as shown in Figures 4.21b and 4.21c. The latter figure shows a passive reference with an unknown "Cell Used" portion because of the double asterisk in the instruction at location B. During the Data Processing Phase, the active reference to the instruction

Figure 4.19 - The Reference List Fntries of the CLA Instruction

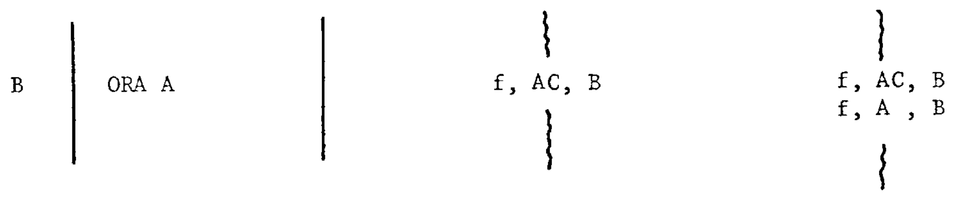


a. The Program

b. The Active Reference List

c. The Passive Reference List

Figure 4. 20 - The Reference List Entries of the ORA Instruction

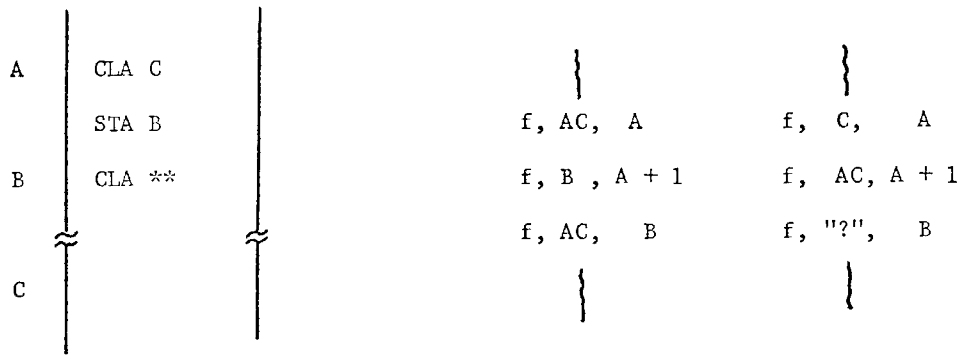


a. The Program

b. The Active Reference List

c. The Passive Reference List

Figure 4.21 - The Reference List Entries for a Changed Address Instruction

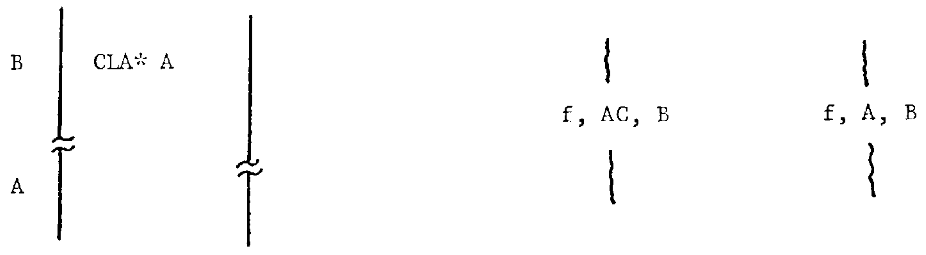


a. The Program

b. The Active Reference List

c. The Passive Reference List

Figure 4.22 - The Reference List Entries for an Indirectly Addressed Instruction



a. The Program

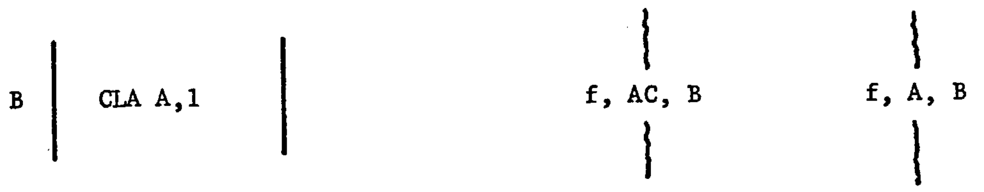
b. The Active Reference List

c. The Passive Reference List

at location B by the STA instruction is detected. Thus, the analysis program must find the functional expression for the "Cell Used" by location B before it can find the functional expression for the information processing performed by that instruction. This order of functional determination can be initiated by setting a special flag in the "f" portion of the passive reference entries for the modified or changed instruction. Therefore, the latest reference searching procedure can detect the changed instruction flag and can initiate the proper search procedure.

The second example of a dependent reference instruction is the indirectly addressed instruction. Figure 4.22a shows how an indirectly addressed instruction might occur in a program. Because the indirect address asterisk can be detected while the instruction line is being decoded, the Data Generation Phase knows it has an indirectly addressed instruction and can generate the proper Reference List entries. For such an instruction, the analysis program must first find the functional expression for the address portion of the cell specified by the operand address of the instruction before it can determine the functional expression for the information processing performed by the instruction. In Figure 4.22a at location B, the address portion of cell A must be found before the contents of the AC can be determined. This order of functional generation can be initiated by constructing a passive reference entry whose "f" portion indicates an indirect instruction. Therefore, the latest reference searching procedure can detect the indirect flag and initiate the proper search procedure.

Figure 4.23 - The Reference List Entries for a Tagged Instruction



a. The Program

b. The Active Reference List

c. The Passive Reference List

The third example of a dependent reference instruction is the tagged instruction. Figure 4.23a shows how a tagged address instruction might occur in a program. Again, because the presence of a tag can be detected while the instruction line is being decoded, the Data Generation Phase knows it has a tagged instruction and can generate the proper Reference List entries. For such an instruction, the analysis program must first find the functional expression for the index register specified by the tag before it can determine the functional expression for the information processing of the tagged instruction. In Figure 4.23a at location B, the contents of the index register must be found before the contents of the AC can be determined. This order of functional generation can be initiated by constructing a passive reference entry whose "f" portion indicates a tagged instruction and an index number. Therefore, the latest reference searching procedure can detect the tag flag and initiate the proper search procedure.

4.4.3 Finding the Latest Reference Sets

In the previous section the motivation and technique for constructing the necessary Reference List entries were described. The function of these entries is to insure that the analysis program can decide the sequence in which it needs to determine the information processing of the program. The purpose of this section is to explain how the analysis program decides which latest reference searches are required and how the program performs those searches.

After all the Reference List entries have been made by the Data Gathering and Data Processing Phases, the Data Reduction Phase associates each Reference List entry with the program block in which the reference occurs. First, the Reference Lists are sorted on their "Instruction Cell" portion to place them in the same sequence as the Topology Table. Second, the Active and Passive Reference Lists are scanned, and their entries placed into the Active and Passive Reference Tables. Figure 4.24a shows an example program block and its instructions. Figures 4.24b and 4.24c show the Sorted Active and Sorted Passive Reference Lists for the example block. Figure 4.24d shows how the entries of those two lists would be placed in the Reference Table.

The latest reference searching procedure must find all the latest references for each Passive Reference Table entry. The search procedure should be performed iteratively but yet be able to decide the search sequence and handle any program topology, such as loops or parallel paths. The search sequence for each passive reference is dictated by the special

Figure 4.24 - Topology Table with Active and Passive Entries

```
 9   TZE    20
10   CLA   100
11   ADD   101
12   STO   102
13   TZE    30
14
```

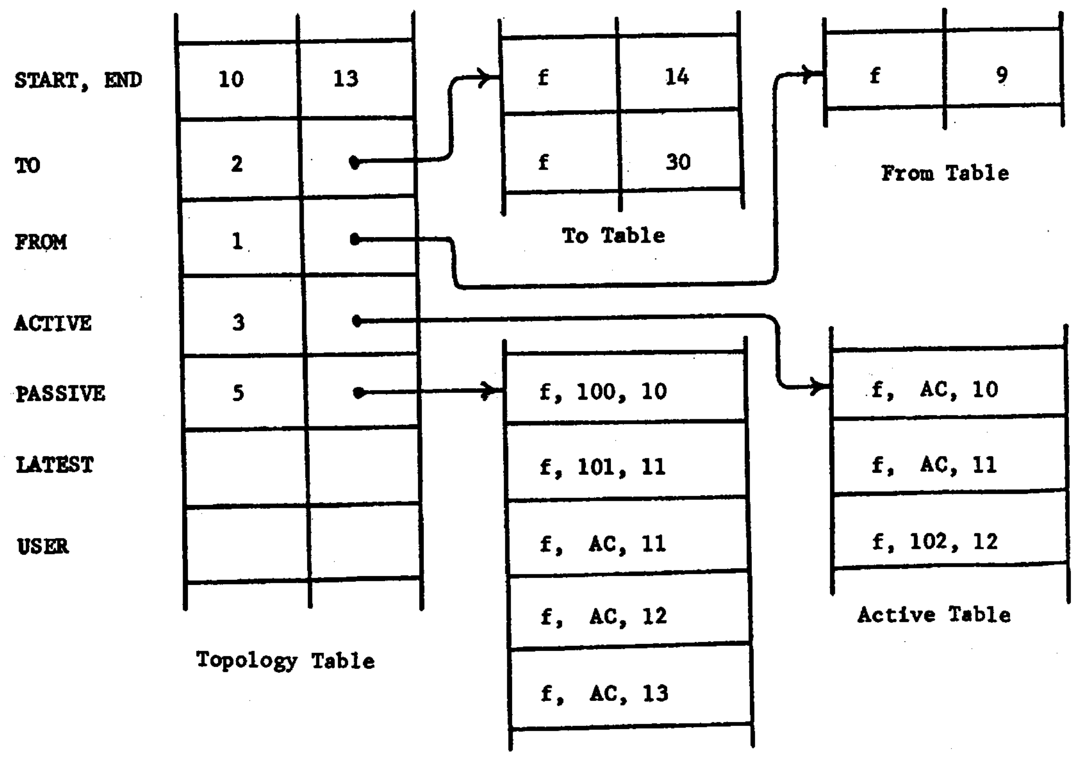
a. The Program

```
f,  AC, 10
f,  AC, 11
f, 102, 12
```

b. Sorted Active List

```
f, 100, 10
f, 101, 11
f,  AC, 11
f,  AC, 12
f,  AC, 13
```

c. Sorted Passive List

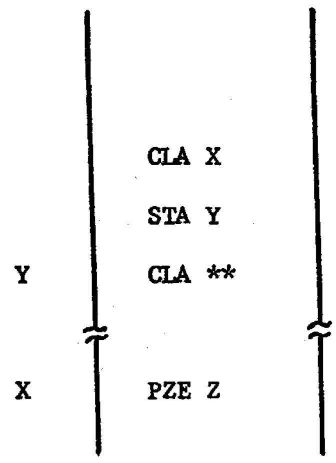


d. The Tables

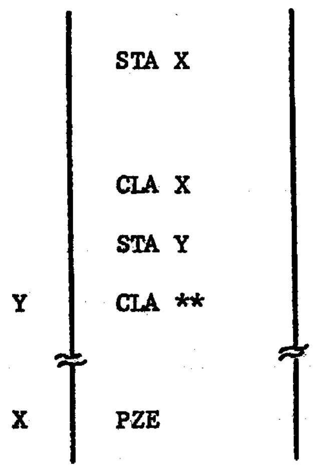
flags set in the "f" portion of the passive reference entry. The search procedure involves searching back from each passive reference entry on all control paths until each path is terminated by a matching active reference entry; the initial passive reference entry; or a previously searched block. A matching active reference is an active reference whose "Cell Changed" portion matches the "Cell Used" portion of the initial passive reference. The "f" portion of each passive reference entry states which cell bits are used by the passive reference; the "f" portion of each active reference entry states which cell bits are changed by the active reference. Thus, the latest reference searching procedure is capable of detecting partial bit matches and can continue searching along a path until all "Cell Used" bits have been matched by "Cell Changed" bits.

If the "f" portion of the passive reference indicates a changed address, the latest references for the changed address must first be determined. If the first search finds only one latest reference and determines that the latest reference stores a constant into the changed address, a second search can be performed to find the latest references for the cell specified by the previously determined constant. Figure 4.25a shows an example of a first search resulting in a constant. The first latest reference search on the changed address instruction at location Y indicates its true "Cell Used" is location Z. The second latest reference search can be performed as if location Y was a CLA Z instruction. On the other hand, if the first changed address search finds one or more variable expressions for the changed address, no accurate second search can be performed during the first iteration. Figure 4.25b shows an example of a first search resulting in a variable. The first latest reference search on location Y indicates that the address portion of Y comes from the address portion of X. However, the address portion of X is a variable because of the STA X instruction. Therefore, no second search can be performed during the first iteration. Only an approximate expression of the form, AC = C(a/X), can be produced as output for location Y after the first iteration. In Iverson Notation (7), a/X indicates the address portion of location X; and C(x) means "the contents of location x".

Figure 4.25 - Programs with Changed Addresses



a. A Constant
Changed Address

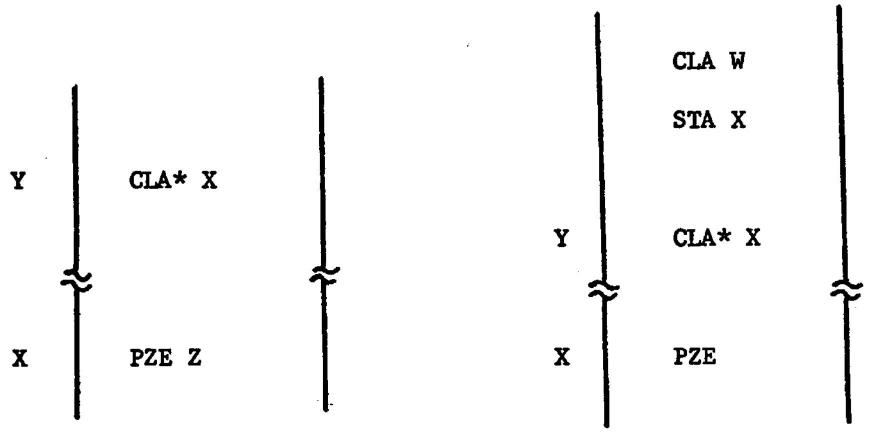


b. A Variable
Changed Address

If the "f" portion of the passive reference indicates an indirect address, the address portion of the cell specified by the address of the indirect instruction must first be determined. If the first search finds only one latest reference to that cell and determines that the latest reference stores a constant into that cell, a second search can be performed to find the latest references for the cell specified by the previously determined constant. Figure 4.26a shows an example of a first search resulting in a constant. The first latest reference search on the address portion of X indicates that it is a constant, Z. The second latest reference search is performed as if location Y was a CLA Z instruction. On the other hand, if the first search determines that one or more variable expressions are stored into the address portion of the location specified by the indirect instruction, no accurate second search can be performed during the first iteration. Figure 4.26b shows an example of a first search resulting in a variable. The first latest reference search on location Y indicates that the address portion of X comes from the address portion of W. Thus the address portion of X is a variable because of the STA X instruction. Therefore, no second search can be performed during the first iteration. Only an approximate expression of the form, AC = C(a/W), can be produced as output for location Y after the first iteration.

If the "f" portion of the passive reference indicates a tagged reference, latest reference searches are performed on both the "Cell Used" and the index register of the tagged instruction. The first search

Figure 4.26 - Programs with Indirect Addresses



a. A Constant Indirect Address

b. A Variable Indirect Address

determines the latest references for the "Cell Used", i.e. which instructions could have last made entries into the table headed by the "Cell Used" location. The second search determines the latest references for the index register used by the tagged instruction, i.e. which instructions last modified the index register. Two searches are performed because there is little chance that the index register is a constant and that the exact "Cell Used" can be determined by the first iteration.

Finally, if the "f" portion of the passive reference entry does not contain any special latest reference search flags, the latest reference search is performed directly on the "Cell Used" portion of the passive reference.

4.4.4 Saving the Latest Reference Information

After the latest reference set of a passive reference has been determined, the latest reference information must be saved in a data structure which permits the generation of function expressions for each instruction and the transmission of those expressions to other instructions. The purpose of this section is to discuss temporary list structures for latest reference information and the final Latest Reference Tables which fulfill the above requirements.

If the data structure of the temporary Latest Reference Lists is to conform with the general solution philosophy discussed earlier, the structure must permit individual entries to be added as required but yet allow all entries to be processed as a group. These characteristics can be incorporated into two lists, the Latest Reference List and the User List. The Latest Reference List contains latest reference entries which remember the locations of all latest references for each passive reference. The User Reference List contains user entries which remember the locations of the passive references which will require the functional expressions produced by each active reference. The format of the list entries is shown in Figure 4.27.

The Latest Reference List entries are divided into three parts. The first portion is the "Latest Reference Cell" and is the "Instruction Cell" of the Active Reference Table entry which produced the match during the latest reference search. The second portion is the "Cell Used" and is the same "Cell Used" as in the passive entry which initiated the latest

Figure 4.27 - The Latest Reference and User List Formats

LATEST REFERENCE CELL	CELL USED	PASSIVE INSTRUCTION CELL

a. The Latest Reference List

LATEST REFERENCE POINTER	CELL CHANGED	ACTIVE INSTRUCTION CELL

b. The User List

reference search. The third portion is the "Instruction Cell" of the passive entry and is used to identify which Latest Reference List entries are associated with each Passive Reference Table entry.

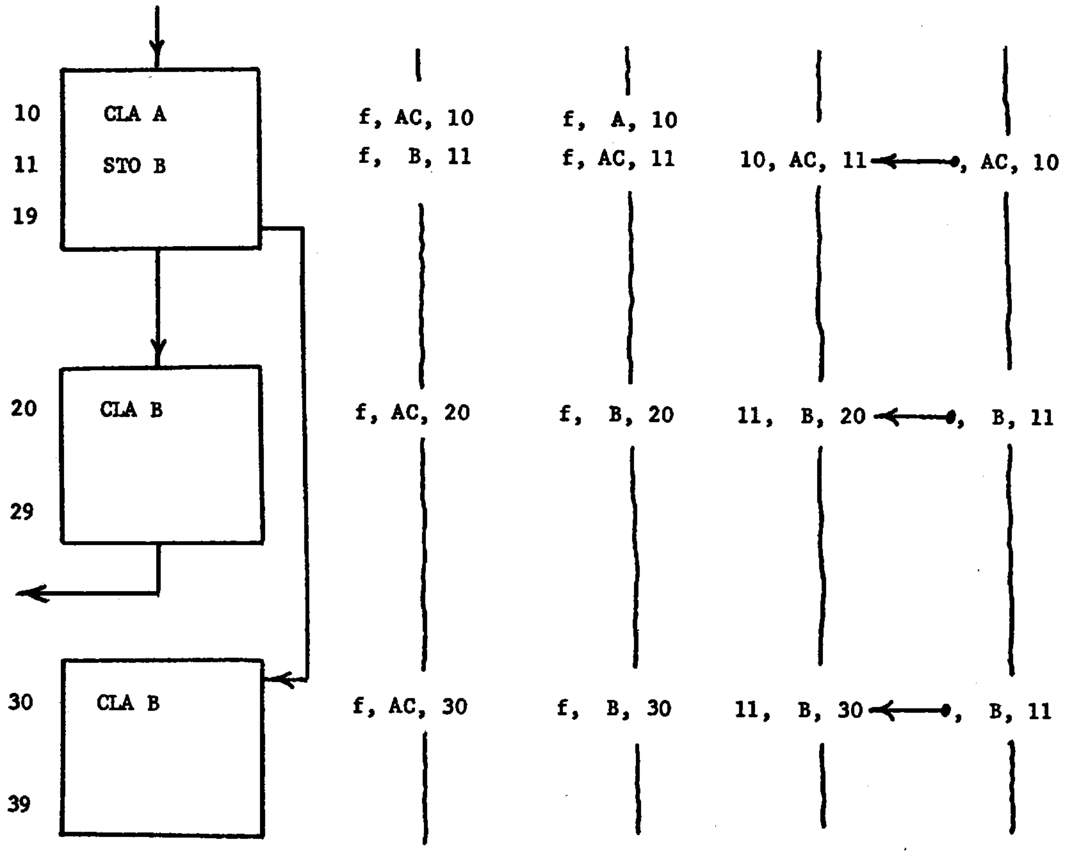
The User List entries are also divided into three parts. The first portion is the "Latest Reference Pointer" and points to the location which contains its Latest Reference mate. The second portion is the "Cell Changed" of the Active Reference Table entry which produced the match. The third portion is the "Instruction Cell" of that Active

Reference Table entry and is used to identify which User List entries are associated with each Active Reference Table entry.

As each latest reference search match is found, one entry is added to the Latest Reference List and the User List. The absence of Latest Reference List entries for a passive reference indicates no latest references were found. Figure 4.28 shows the Latest Reference List and User List entries that would result for a program where a functional expression is needed by two instructions elsewhere in the program. The functional expression generated by location 11 is needed at locations 20 and 30. At location 20 there is a passive reference to location B which has one latest reference at location 11. Thus, a single latest reference entry is added to the Latest Reference List showing the latest reference information, and one user entry is added to the User List. Likewise, at location 30 there is a passive reference to location B which has one latest reference at location 11.

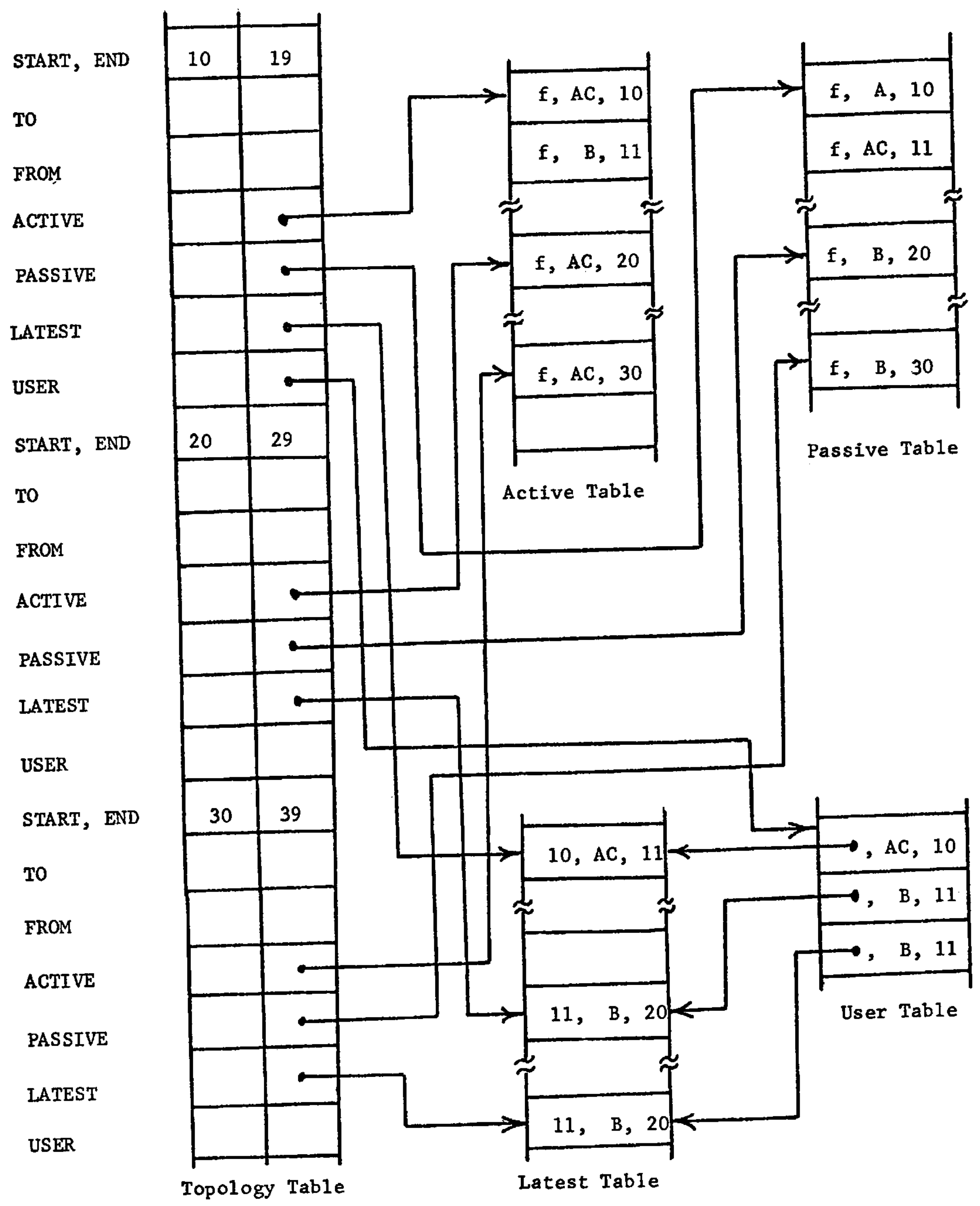
The temporary lists are transformed into the final Latest Reference Table and the User Table by associating each list entry with the program block in which it occurs. The Latest Reference List is sorted on its "Passive Instruction Cell" portion while the User List is sorted on its "Active Instruction Cell" portion. The resulting list entries are associated with the blocks in which they occur by scanning the ordered lists and constructing the "Latest" and "User" entries in the Topology Table. Figure 4.29 shows the resulting tables for the example shown in Figure 4.28.

Figure 4.28 - A Program where Symbolic Results are needed at Two Later Points in the Program



a. The Program b. Active Table c. Passive Table d. Latest List e. User List

Figure 4.29 - The Final Data Flow Tables



In summary, each Latest Reference Table entry points from a passive reference back to an active reference which is a member of the passive reference's latest reference set. Each User Table entry points from an active reference forward to a passive reference which will need the functional expression generated by the active reference.

4.4.5 Constructing the Functional Expressions

A functional expression is generated for each active reference entry. The instruction operation code retained in the "f" portion of the active reference entry dictates its functional expression format. As the construction of a new expression begins, the expression format is found by extracting the instruction operation code from the active reference "f" portion and using the code as a table lookup pointer for the Format Table. The Format Table entry for each instruction indicates the functional expression format for each of the active references of the instruction. The table entry includes the number of entries to be expected in the Active and Passive Reference Tables and the operator symbols to be used in constructing the functional expressions. Whenever possible, a latest reference expression already generated for a previous active reference is substituted for each passive reference in the new active reference expression. Now, functional expression construction is discussed in detail using the program in Figures 4.28 and 4.29 as an example. First, the discussion will explain how the functional expressions, AC = A and B = A, are constructed for locations 10 and 11. Second, the discussion will outline how the expression, B = A, is transmitted from location 11 to locations 20 and 30.

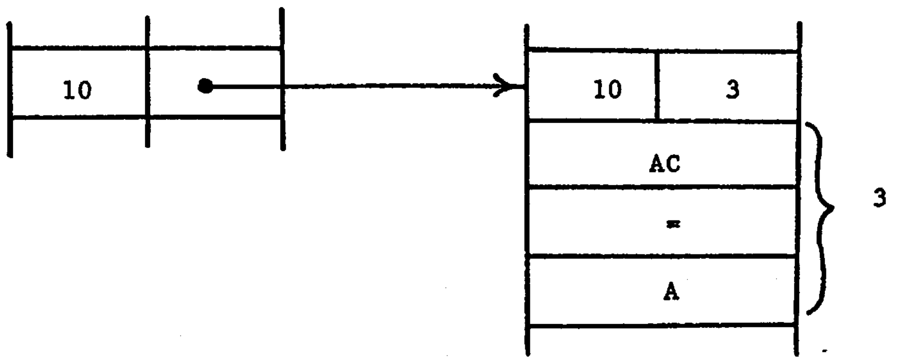
The Active Reference Table entry for location 10 in Figure 4.29 is "f,AC,10" where "f" indicates a CLA instruction. The Format Table entry for a CLA instruction indicates an expression format of:

"CELL CHANGED" = "LATEST EXPRESSION" (or "CELL USED" if no latest expression)

The Passive Reference Table entry for the CLA instruction is found by finding a matching "Instruction Cell" value of 10. In this case the passive entry is "f,A,10". The Latest Reference Table entries for this passive reference are found by matching the two right-hand portions of each entry. In this case, there are no latest reference entries for location 10. Thus, the functional expression for location 10 is AC = A.

The new functional expression is held for final output processing by adding it to the functional Output List. Figure 4.30 outlines the data structure of the Output List. The final output processing will need to sequence the functional expression strings according to instruction location. To facilitate this resequencing, a message pointer is constructed and added to the Message Pointer List. The left half of each Message Pointer List entry indicates the instruction location to which its expression applies, and the right side points to the expression itself. Thus, the Output List expressions are ordered by sorting the single

Figure 4.30 - The Functional Expressions on the Output List



a. Message Pointer List b. Output List

entry Message Pointer List instead of the variable length entry Output List.

Once the functional expression is constructed, the User Table must be checked to see if any instructions further on in the program need this expression. The user entry for the active reference entry is found by matching the two right-hand portions of each entry. In this case there is one user entry, "Pointer,AC,10". This user entry states that the latest reference entry at the end of the pointer wants to know the just derived expression for the AC. The analysis program follows the pointer to its Latest Reference Table entry mate, "10,AC,11". Once the entry is found, its "Latest Reference Cell" portion is replaced by a pointer to the just constructed functional expression on the Output List. Also, the latest reference entry is flagged as having an expression pointer. Now, the latest reference entry at location 11 knows where the functional expression for the AC can be found, i.e. the functional expression has been transmitted from location 10 to location 11.

The active entry for location 11 in Figure 4.29 is "f,B,11" where "f" indicates a STO instruction. The Format Table entry for the STO instruction indicates an expression format of:

"CELL CHANGED" = "LATEST EXPRESSION" (or "CELL USED" if no latest expression)

The Passive Reference Table entry for the STO instruction is "f,AC,11". One matching latest reference entry, "Expression Pointer,AC,11" is found. This is the latest reference entry that was found by following the pointer of the previous user entry. The functional expression for location 11 is constructed by first adding the "Cell Changed" to the Output List. In this case, the "Cell Changed" is B. Next, the symbolic equal sign is added to the Output List. Finally, the expression pointer of the latest reference entry is followed to its functional expression, AC = A. The expression is scanned until the equal sign is found, and the remaining entries after the equal sign are copied onto the Output List. Thus, the expression, B = A, is generated for location 11.

Finally, two identical user entries (Pointer,B,11 and Pointer,B,11) are found for location 11 in Figure 4.29. Each of the entry pointers is followed to its latest reference mate. Each "Latest Reference Cell" portion is replaced by a pointer to the just derived functional expression, B = A, on the Output List; and the latest reference entry is flagged as having an expression pointer. Thus, when locations 20 and 30 are reached, the functional expression for location B is available.

CHAPTER 5

AUTOMATIC PROGRAM ANALYSIS EXAMPLES

In the previous chapter the approximation procedures used by the first iteration to bootstrap itself through the control flow - data flow interaction cycle were shown. This outline described the data acquisition and data processing sequence and showed the use of intermediate data flow analysis results to improve control flow approximations and vice versa. In addition, a detailed presentation described how the control and data flow steps handled the dependent instructions.

This chapter displays the results of applying the existing automatic analysis system to example programs. First, the layout and symbols of the output flowcharts are explained. Second, flowcharts of programs containing dependent instructions are described. Third, flowcharts of programs containing other analysis problems are presented. All output examples were automatically produced on-line by an IBM 1052 printer keyboard connected to the Project MAC IBM 7094 time-shared computer (2). Because the IBM 1052 printer does not normally contain the complete Iverson Notation character set (7), character substitutions have been made.

5.1 THE FLOWCHART FORMATS

The analysis program should display its results in a form suitable for human use. Because the flowchart has become a standard vehicle for program documentation, it is also used here. Currently, the analysis system has two levels of flowchart detail: the Topological Flowchart and the Detailed Flowchart. The Topological Flowchart presents the control flow of a program by displaying its block execution sequence. The Detailed Flowchart exhibits both control and data flows by displaying the block execution sequence, the functional expressions, and pertinent cross reference information. One example of each flowchart type is discussed in detail so that only the highlights of later examples need to be explained.

5.1.1 The Topological Flowchart

Figure 5.1 exhibits an example of a Topological Flowchart. The program always starts at Block 1. The asterisks represent the instructions contained within a block. The number at the upper left of each block is its Block Number. The dots represent control flow paths. The block inputs always enter at the top of the block; the outputs always exit at the bottom. No attempt has been made to minimize line crossings by rearranging blocks. Now, the interpretation of the flowchart symbols of Figure 5.1 is given.

Block 1 is the starting block and exits to either Block 2 or Block 4.

Figure 5.1 - A Topological Flowchart

```
 1
      *
      *
      *
      *)................
      .                .
 2    .                .
      *                .
      *                .
      *                .
      *).............................................................E4
                       .
 3                     .
      *                .
      *                .
      *                .
      *                .
                       .
 4                     .
      *(.............................................................E2
      *
      *
      *).............................................................I5

 5
      *(.............................................................I4
      *
      *
      *).............................................................NR

 6
      *(.............................................................IE4
      *
      *
      *
      .
 7    .
      *(........
      *        .
      *        .
      *)........
      .
10    .
      *
      *
      *
      *).............................................................IR

11
      *
      *
      *
      *
```

<u>Block 2</u> can be reached from Block 1 and has an "E4" exit. The "E" designates an exit to an external subroutine; the "4" indicates that the external subroutine returns control to Block 4.

<u>Block 3</u> is unreachable and has no exits. Because it follows an external subroutine exit, it is probably a subroutine calling sequence.

<u>Block 4</u> can be reached from Block 1 and "E2". The "E" signifies an entry from an external subroutine; the "2" denotes that the external subroutine is called by Block 2. Block 4 has an "I5" exit. The "I" specifies an exit to an internal subroutine; the "5" reveals that the internal subroutine returns control to Block 5.

<u>Block 5</u> can be reached by an "I4" entry which denotes a return from an internal subroutine called at Block 4. Block 5 has a "NR" exit which indicates a non-returning external subroutine call.

<u>Block 6</u> can be reached by an "IE4" entry where the "IE" designates an internal subroutine entry and the "4" reveals that the subroutine is called by Block 4. Block 6 exits to Block 7.

<u>Block 7</u> can be reached from itself or Block 6 and exits to Block 10 or to itself.

<u>Block 10</u> can be reached from Block 7 and exits via an "IR". The "IR" specifies an internal subroutine return, such as a TRA 1,4.

<u>Block 11</u> appears to be a data and storage area.

5.1.2 The Detailed Flowchart

Figure 5.2 shows an example of a Detailed Flowchart. The first three lines and the last line on the flowchart page were produced by the time-sharing monitor as it prepared the final analysis phase for execution. The left side of the output exhibits the original symbolic source instructions and their assigned core locations; the right side displays the flowchart box outlines, interconnections, and functional expressions. The Block Numbers are shown above each block. The starting and ending core locations of each block are shown on the left side of the block. The block inputs always enter at the top or upper right of the block; the outputs always exit at the bottom or lower right. The numbers to the right of the entering or exiting dots are Block Numbers to which or from which control is transferred. The expressions inside the flowchart boxes are the functional expressions. The expressions outside the boxes are cross reference expressions preceded by the location number of the instruction which generated the expression. Now, the flowchart symbols of Figure 5.2 are explained.

Block 1 is the starting block and exits to either Block 2 or Block 3. The first instruction of Block 1 is at location 1; the last is at location 3. At location 1 the contents of location V are placed into the accumulator. At location 2 the contents of location V are moved to location W. The cross reference expression at the right of location 2 states that the AC was changed to the contents of V at location 1. The line for location 3 is blank because of unprogrammed subroutines. (See Appendix 1 for missing subroutine information.) If the

Figure 5.2 - A Detailed Flowchart

```
r run5 000000
W 1907.0
EXECUTION.
                                     1
                                ************************
                             1  *                      *
01       CLA      V             *     AC=V             *
02       STO      W             *     W=V              *      1 AC=V
03       TZE      A1            *                      *
                             3  *                      *)...  3
                                ************************
                                       .
                                       .
                                       .
                                     2 .
                                ************************
                             4  *                      *
04       CLA      X             *     AC=1             *
05       STO      Y             *     Y=1              *      4 AC=1
                             5  *                      *
                                ************************
                                       .
                                       .
                                       .
                                     3 .
                                ************************
                             6  *                      *(...  1
06 A1    CLA      W             *     AC=V             *      2 W=V
07       STA      Z             *     A/Z=A/V          *      6 AC=V
10       TSX      $EXIT,4       *                      *
                            10  *                      *)...  NR
                                ************************


                                     4
                                ************************
                            11  *                      *
11 V     BSS      1             *                      *
12 W     BSS      1             *                      *
13 X     OCT      1             *                      *
14 Y     BSS      1             *                      *
15 Z     BSS      1             *                      *
                            15  *                      *
                                ************************


R 4.750+3.000
```

programming was complete, the line would show V:0; and the cross reference expression would state that the AC was V at location 1.

Block 2 can be reached from Block 1 and exits directly to Block 3. The first instruction of Block 2 is at location 4; the last is at location 5. At location 4 the contents of X are placed into the AC. Since the contents of X are constant, the symbol X is replaced by its constant value, 1, in the functional expression for location 4. At location 5 the constant, 1, is stored into location Y.

Block 3 can be reached from either Block 1 or Block 3 and has a non-returning exit. The first instruction of Block 3 is at location 6; the last is at location 10. At location 6, the contents of location W, which are now the contents of V, are placed into the AC. The cross reference expression states that the contents of V were placed in W at location 2. At location 7 the address portion of Z is replaced by the address portion of V.

Block 4 is unreachable. It begins at location 11 and ends at location 15. Block 4 is empty because it contains data and storage locations.

5.2 FLOWCHARTS CONTAINING DEPENDENT INSTRUCTIONS

The purpose of this section is to show examples of automatically produced flowcharts for programs containing dependent instructions. The example programs have been kept short so as to spotlight the individual dependent instructions. Instead of being viewed as programs in themselves, the examples might be thought of as being imbedded in larger programs. Since both the Topological and Detailed Flowchart conventions have been discussed, only the pertinent results are explained in the following examples.

5.2.1 The Transfer Switch

Figure 5.3 shows the first example of a program containing a transfer switch. At location 3 a passive reference is made to location 5 which contains a transfer instruction, TRA END. At location 4 the transfer switch is stored into location 3, i.e. A1. Thus, Block 1 is terminated by a transfer switch, and control paths are generated from Block 1 to Blocks 2 and 4. Note, the analysis program found that the transfer instruction at location 5 can be executed in that location. Therefore, Block 2 is terminated at location 5.

Figure 5.4 shows a second example of a program containing a transfer switch. The analysis program found that the transfer instruction at location 10 is not executed in its original core location. Thus, location 10 is included in Block 3 as data.

5.2.2 The Subroutine Call and Return

Figure 5.5 shows an example of a program containing subroutine calls and returns. At location 4 the internal subroutine, "IN", is called. The analysis program detects the internal subroutine entry point at

Figure 5.3 - A Transfer Switch Executed in its Original Location

```
r run5 000000
W 1423.4
EXECUTION.
                                    1
                              ************************
                            1 *                      *
01     CLA     X              *     AC=1             *
02     STO     Y              *     Y=1              *       1 AC=1
03 A1  CAL     A              *     AC=TRA END       *
                            3 *                      *)...   4
                              ************************
                                           .
                                           .
                                           .
                                    2      .
                              ************************
                            4 *                      *
04     SLW     A1             *     A1=TRA END       *       3 AC=TRA END
05 A   TRA     END            *                      *
                            5 *                      *)...   4
                              ************************


                                    3
                              ************************
                            6 *                      *
06 X   OCT     1              *                      *
07 Y   BSS     1              *                      *
                            7 *                      *
                              ************************


                                    4
                              ************************
                           10 *                      *(...   1,2
10 END TSX     $EXIT,4        *                      *
                           10 *                      *)...   NR
                              ************************


R 5.283+2.833
```

Figure 5.4 - A Transfer Switch Not Executed in its Original Location

```
r run5 000000
W 1614.9
EXECUTION.
                                           1
                                  ************************
                               1  *                      *
01      CLA      X                *     AC=1             *
02      STO      Y                *     Y=1              *      1 AC=1
03 A1   CAL      A                *     AC=TRA END       *
                               3  *                      *)...  4
                                  ************************
                                                .
                                                .
                                                .
                                           2    .
                                  ************************
                               4  *                      *
04      SLW      A1               *     A1=TRA END       *      3 AC=TRA END
05      TRA      END              *                      *
                               5  *                      *)...  4
                                  ************************


                                           3
                                  ************************
                               6  *                      *
06 X    OCT      1                *                      *
07 Y    BSS      1                *                      *
10 A    TRA      END              *                      *
                              10  *                      *
                                  ************************


                                           4
                                  ************************
                              11  *                      *(...  1,2
11 END TSX      $EXIT,4           *                      *
                              11  *                      *)...  NR
                                  ************************


R 4.483+3.150
```

Figure 5.5 - Subroutine Calls and Returns

```
r run5 000000
W 1607.0
EXECUTION.
                                                  1
                                    *************************
                                  2 *                       *
02       CLA      A                 *     AC=1              *
03       STA      X                 *     A/X=A/1           *      2 AC=1
04       TSX      IN,4              *                       *
                                  4 *                       *)...  13
                                    *************************


                                                  2
                                    *************************
                                  5 *                       *
05 X     PZE      0                 *                       *
                                  5 *                       *
                                    *************************


                                                  3
                                    *************************
                                  6 *                       *(...  11
06       TSX      $EXTERN,4         *                       *
                                  6 *                       *)...  E4
                                    *************************


                                                  4
                                    *************************
                                  7 *                       *(...  E3
07       CLA      A                 *     AC=1              *
10       STO      B                 *     B=1               *      7 AC=1
11       TSX      $EXIT,4           *                       *
                                 11 *                       *)...  NR
                                    *************************


                                                  5
                                    *************************
                                 12 *                       *(...  IE1
12 IN    CLA      A                 *     AC=1              *
13       STO      B                 *     B=1               *      12 AC=1
14       TRA      2,4               *                       *
                                 14 *                       *)...  IR
                                    *************************


                                                  6
                                    *************************
                                 15 *                       *
15 A     OCT      1                 *                       *
16 B     BSS      1                 *                       *
                                 16 *                       *
                                    *************************
```

location 12 and the return at location 14. Because the subroutine returns via a TRA 2,4, location 5 is assumed to be a single instruction calling sequence. External subroutines are called at locations 6 and 11.

5.2.3 The Calculated Transfer

Figure 5.6a shows the Detailed Flowchart of a program containing a tagged transfer instruction. The analysis program assumes that Blocks 2 and 3 can be reached from Block 1. The "L4" and "L5" entries to Block 1 indicate that they close control loops from Blocks 4 and 5. Likewise, the "L1" exits from Blocks 4 and 5 specify loops back to Block 1. Figure 5.6b shows the Topology Flowchart for the same program.

5.2.4 The Changed Address

Figure 5.7 shows the first example of a program containing a changed address instruction. At location 1 the address portion of W, the constant 7, is stored into the address portion of location 2. When the instruction at location 2 is executed, it is a CLA 7. Therefore, the contents of location 7 or Z are placed into the AC at location 2. This address change can be traced during the first iteration because a single constant was used as the new address for the changed address instruction.

Figure 5.8 shows the second example of a program containing a changed address. The address portion of location Y is used as the new address at location 4. In this case location Y appears to be a variable during

Figure 5.6a - A Calculated Transfer

```
r run5 000000
W 1730.8
EXECUTION.
                                   1
                              ************************
                           0  *                      *(...  L4,L5
00 A1   LXA     A,1           *     IX1=A/1          *
01      TRA     *,1           *                      *
                           1  *                      *)...  3
                              ************************
                                          .
                                          .
                                          .
                                   2      .
                              ************************
                           2  *                      *
02      TRA     B             *                      *
                           2  *                      *)...  4
                              ************************


                                   3
                              ************************
                           3  *                      *(...  1
03      TRA     C             *                      *
                           3  *                      *)...  5
                              ************************


                                   4
                              ************************
                           4  *                      *(...  2
04 B    CLA     D             *     AC=1             *
05      STO     F             *     F=1              *      4 AC=1
06      TRA     A1            *                      *
                           6  *                      *)...  L1
                              ************************


                                   5
                              ************************
                           7  *                      *(...  3
07 C    CLA     E             *     AC=2             *
10      STO     F             *     F=2              *      7 AC=2
11      TRA     A1            *                      *
                          11  *                      *)...  L1
                              ************************


                                   6
                              ************************
                          12  *                      *
12 A    OCT     1             *                      *
13 D    OCT     1             *                      *
14 E    OCT     2             *                      *
15 F    BSS     1             *                      *
                          15  *                      *
                              ************************
```

Figure 5.6b - The Topology Flowchart of the Program in Figure 5.6a

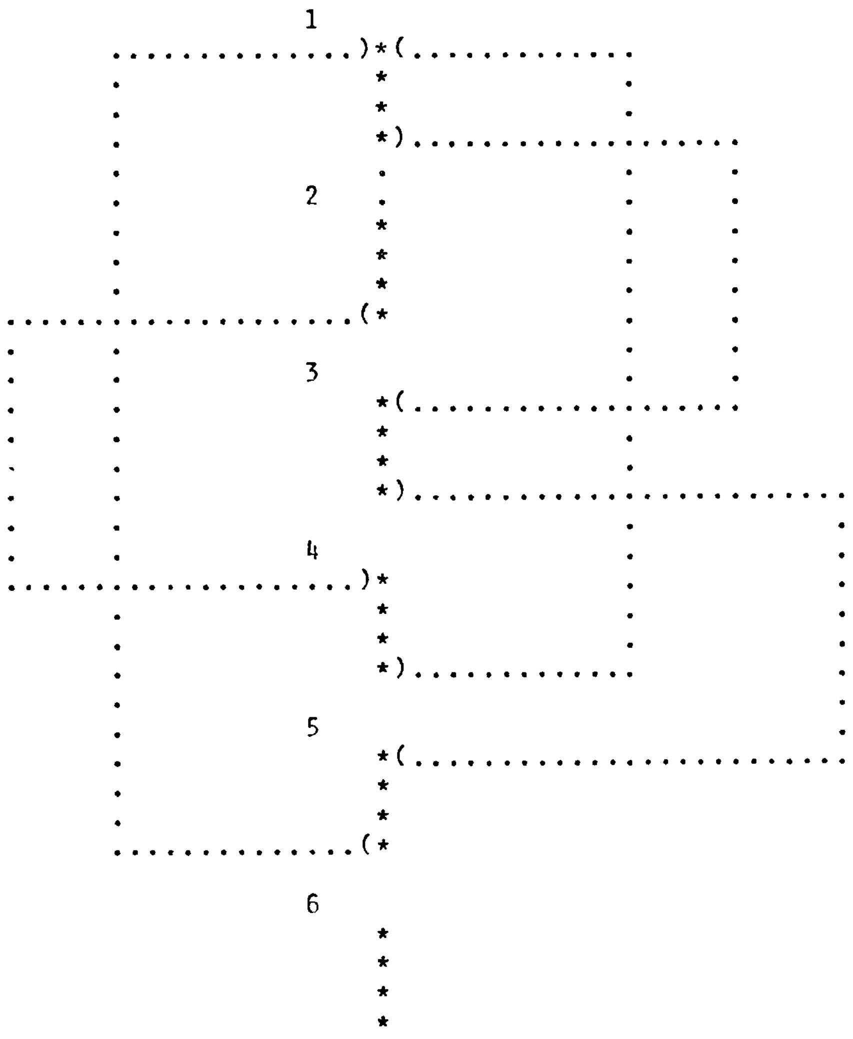


Figure 5.7 - A Changed Address Using a Single Constant

```
r run5 000000
W 1741.5
EXECUTION.
                                                    1
                                              ************************
                                         0    *                      *(...   L1
00 A1   CLA     W                             *     AC=7             *
01      STA     X                             *     A/X=A/7          *       0 AC=7
02 X    CLA     **                            *     AC=Z             *       1 A/X=A/7
03      STO     Y                             *     Y=Z              *       2 AC=Z
04      TRA     A1                            *                      *
                                         4    *                      *)...   L1
                                              ************************


                                                    2
                                              ************************
                                         5    *                      *
05 W    PZE     Z                             *                      *
06 Y    BSS     1                             *                      *
07 Z    OCT     1                             *                      *
                                         7    *                      *
                                              ************************


R 3.583+3.400
```

Figure 5.8 - A Changed Address Using a Single Variable

```
r run5 000000
W 1801.0
EXECUTION.
                                              1
                                    ***********************
                                 0  *                     *(...   L1
00 A1   CLA     X                   *     AC=10           *
01      STA     Y                   *     A/Y=A/10        *       0 AC=10
02      CLA     Y                   *     AC=A/10         *       1 A/Y=A/10
03      STA     B                   *     A/B=A/A/10      *       2 AC=A/10
04 B    CLA     **                  *     AC=C(A/A/10)    *       3 A/B=A/A/10
05      STO     C                   *     C=AC            *       4 AC=C(A/A/10)
06      TRA     A1                  *                     *
                                 6  *                     *)...   L1
                                    ***********************


                                              2
                                    ***********************
                                 7  *                     *
07 C    BSS     1                   *                     *
10 D    OCT     1                   *                     *
11 X    PZE     D                   *                     *
12 Y    PZE     0                   *                     *
                                12  *                     *
                                    ***********************


R 4.200+2.783
```

the first iteration because of the STA Y instruction at location 1. Because the analysis program believes that the new address of location 4 is also a variable, the functional expression for that location states that the AC contains the contents of the location whose address is 10 or D. (In Iverson Notation, A/A/10 = A/10.)

Figure 5.9 shows a third example of a program containing a changed address. The instruction at location 5 can have its address changed from either location 1 or location 4. The cross reference expressions at location 5 show the two possible values for its new address. If location 1 changes the address, it becomes location 10 or D. If location 4 changes the address, it becomes location 12 or E. Because the address of location 5 can be changed from two possible locations, its functional expression states that the contents of an undetermined location are placed into the AC.

5.2.5 The Indirect Address

Figure 5.10 shows the first example of a program containing an indirectly addressed instruction. The analysis program detects that the address portion of location A is a constant and that location 0 actually is a CLA C instruction. Therefore, location 1 loads the contents of location C into the AC. During the first iteration, the Data Gathering Phase had no reason to generate a passive reference to location C. Thus, the analysis program does not yet know that location C is the constant, 1.

Figure 5.11 shows a second example of a program containing an

Figure 5.9 - A Changed Address Using Two or More Expressions

```
r run5 000000 000000
W 1809.3
EXECUTION.
                                   1
                              ************************
                           0  *                      *(...  L3
00 A1   CLA     X             *     AC=10            *
01      STA     B             *     A/B=A/10         *      0 AC=10
02      TNZ     B             *                      *
                           2  *                      *)...  3
                              ************************
                                          .
                                          .
                                          .
                                   2      .
                              ************************
                           3  *                      *
03      CLA     Y'            *     AC=12            *
04      STA     B             *     A/B=A/12         *      3 AC=12
                           4  *                      *
                              ************************
                                          .
                                          .
                                          .
                                   3      .
                              ************************
                           5  *                      *(...  1
05 B    CLA     **            *     AC=C(**)         *      1 A/B=A/10
                              *                      *      4 A/B=A/12
06      STO     Z             *     Z=C(**)          *      5 AC=C(**)
07      TRA     A1            *                      *
                           7  *                      *)...  L1
                              ************************


                                   4
                              ************************
                          10  *                      *
10 D    OCT     1             *                      *
11 X    PZE     D             *                      *
12 E    OCT     2             *                      *
13 Y    PZE     E             *                      *
14 Z    BSS     1             *                      *
                          14  *                      *
                              ************************

R 5.566+3.783
```

Figure 5.10 - An Indirect Address Using a Constant

```
r run5 000000
W 2053.7
EXECUTION.
                                   1
                              ************************
                           0  *                      *(...  L1
00 A1  CLA*    A              *     AC=C             *
01     STO     B              *     B=C              *      0 AC=C
02     TRA     A1             *                      *
                           2  *                      *)...  L1
                              ************************


                                   2
                              ************************
                           3  *                      *
03 A   PZE     C              *                      *
04 B   BSS     1              *                      *
05 C   OCT     1              *                      *
                           5  *                      *
                              ************************


R 3.933+2.950
```

Figure 5.11 - An Indirect Address Using a Single Variable

```
r run5 000000
W 1816.8
EXECUTION.
                                   1
                              ************************
                           0  *                      *(...  L1
00 A1  CLA     D              *     AC=6             *
01     STA     E              *     A/E=A/6          *      0 AC=6
02     CLA*    E              *     AC=A/6*          *      1 A/E=A/6
03     STO     B              *     B=A/6*           *      2 AC=A/6*
04     TRA     A1             *                      *
                           4  *                      *)...  L1
                              ************************


                                   2
                              ************************
                           5  *                      *
05 B   BSS     1              *                      *
06 C   OCT     1              *                      *
07 D   PZE     C              *                      *
10 E   PZE     0              *                      *
                          10  *                      *
                              ************************


R 3.433+2.550
```

indirectly addressed instruction. In this case the indirectly addressed location, E, is a variable during the first iteration because of the STA E instruction at location 1. Thus, the functional expression for location 2 states that the AC is loaded indirectly from a location whose address is 6 or C. Once again, the analysis program does not yet know that location C is a constant.

Figure 5.12 shows a third example of a program containing an indirectly addressed instruction. At location 5 the cross reference expressions state that the address portion of the indirectly addressed location, X, can be either 11 or 13. Because the indirectly addressed location can have more than one expression, the functional expression states that the AC is loaded indirectly from X.

5.2.6 The Tagged Address

Figure 5.13 shows an example of a program containing tagged instructions. At location 3 a tagged passive reference is made to location V using index register one. This is stated by the functional expression, AC = V(1). The cross reference expression at location 3 states that index register one was loaded with a constant, 1, at location 2.

5.3 FLOWCHARTS CONTAINING OTHER ANALYSIS PROBLEMS

The purpose of this section is to show examples of automatically produced flowcharts for programs containing general analysis problems which should be handled by any analysis system.

Figure 5.12 - An Indirect Address Using Two or More Expressions

```
r run5 000000
W 1717.6
EXECUTION.
                                    1
                              ************************
                           0  *                      *(...   L3
00 A1  CLA     D              *     AC=11            *
01     STA     X              *     A/X=A/11         *       0 AC=11
02     TNZ     A2             *                      *
                           2  *                      *)...   3
                              ************************
                                           .
                                           .
                                           .
                                    2      .
                              ************************
                           3  *                      *
03     CLA     F              *     AC=13            *
04     STA     X              *     A/X=A/13         *       3 AC=13
                           4  *                      *
                              ************************
                                           .
                                           .
                                           .
                                    3      .
                              ************************
                           5  *                      *(...   1
05 A2  CLA*    X              *     AC=X*            *       1 A/X=A/11
                              *                      *       4 A/X=A/13
06     STO     B              *     B=X*             *       5 AC=X*
07     TRA     A1             *                      *
                           7  *                      *)...   L1
                              ************************


                                    4
                              ************************
                          10  *                      *
10 B   BSS     1              *                      *
11 C   OCT     1              *                      *
12 D   PZE     C              *                      *
13 E   OCT     2              *                      *
14 F   PZE     E              *                      *
15 X   PZE     0              *                      *
                          15  *                      *
                              ************************


R 4.483+3.466
```

Figure 5.13 - A Tagged Instruction

```
r run5 000000
W 1856.0
EXECUTION.
                                           1
                                      ************************
                                   0  *                      *(...   L1
00 A1   CLA      T                    *     AC=1             *
01      STA      U                    *     A/U=A/1          *      0 AC=1
02      LXA      U,1                  *     IX1=A/A/1        *      1 A/U=A/1
03      CLA      V,1                  *     AC=V(1)          *      2 IX1=A/A/1
04      STO      W                    *     W=V(1)           *      3 AC=V(1)
05      LXA      X,2                  *     IX2=A/3          *
06      CLA      Y,2                  *     AC=Y(2)          *      5 IX2=A/3
07      STO      Z                    *     Z=Y(2)           *      6 AC=Y(2)
10      TRA      A1                   *                      *
                                  10  *                      *)...   L1
                                      ************************


                                           2
                                      ************************
                                  11  *                      *
11 T    OCT      1                    *                      *
12 U    BSS      1                    *                      *
13 V    OCT      2                    *                      *
14 W    BSS      1                    *                      *
15 X    OCT      3                    *                      *
16 Y    OCT      4                    *                      *
17 Z    BSS      1                    *                      *
                                  17  *                      *
                                      ************************


R 3.550+2.400
```

5.3.1 The Program Loop

Figure 5.14 shows an example of a program containing a loop. A passive reference is made to location A at location 2. The cross reference expressions indicate that A has two possible values. The first, A = 1, is generated by location 1; the second, A = 2, is generated by location 5. Note that the analysis program detects the second expression even though location 5 is ahead of and in a loop with location 2. Because of the difficulty in displaying the expression, AC = 1 or 2, the symbol A is retained in the functional expression for location 2.

5.3.2 Temporary Storage

Figure 5.15 shows a program which uses temporary storage. The constant value of A is carried through the sequence of loads and stores of the AC until location 12, where D = 1. Likewise, the constant value of W is carried through loads and stores of the MQ until location 13, where Z = 2. Thus, all references to temporary storage are eliminated at locations 12 and 13.

5.3.3 Parallel Latest Reference Search Paths

Figure 5.16 shows a program which contains two parallel latest reference search paths from a passive reference to an active reference. At location 5 there is a passive reference to B. The latest reference

Figure 5.14 - A Program Loop

```
r run5 000000
W 1823.7
EXECUTION.
                                   1
                              ************************
                           0  *                      *
00     CLA     X              *     AC=1             *
01     STO     A              *     A=1              *      0 AC=1
                           1  *                      *
                              ************************
                                          .
                                          .
                                          .
                                   2      .
                              ************************
                           2  *                      *(...  L2
02 A1  CLA     A              *     AC=A             *      1 A=1
                              *                      *      5 A=2
03     STO     B              *     B=A              *      2 AC=A
04     CLA     Y              *     AC=2             *
05     STO     A              *     A=2              *      4 AC=2
06     TRA     A1             *                      *
                           6  *                      *)...  L2
                              ************************



                                   3
                              ************************
                           7  *                      *
07 X   OCT     1              *                      *
10 Y   OCT     2              *                      *
11 A   BSS     1              *                      *
12 B   BSS     1              *                      *
                          12  *                      *
                              ************************


R 4.966+2.783
```

Figure 5.15 - The Elimination of Temporary Storage References

```
r run5 000000
W 1832.4
EXECUTION.
                                   1
                              ************************
                           0  *                      *(...  L1
00 A1  CLA     A              *     AC=1             *
01     LDQ     W              *     MQ=2             *
02     STO     B              *     B=1              *      0 AC=1
03     STQ     X              *     X=2              *      1 MQ=2
04     CLA     B              *     AC=1             *      2 B=1
05     LDQ     X              *     MQ=2             *      3 X=2
06     STO     C              *     C=1              *      4 AC=1
07     STQ     Y              *     Y=2              *      5 MQ=2
10     CLA     C              *     AC=1             *      6 C=1
11     LDQ     Y              *     MQ=2             *      7 Y=2
12     STO     D              *     D=1              *      10 AC=1
13     STQ     Z              *     Z=2              *      11 MQ=2
14     TRA     A1             *                      *
                          14  *                      *)...  L1
                              ************************



                                   2
                              ************************
                          15  *                      *
15 A   OCT     1              *                      *
16 B   PZE                    *                      *
17 C   PZE                    *                      *
20 D   PZE                    *                      *
21 W   OCT     2              *                      *
22 X   PZE                    *                      *
23 Y   PZE                    *                      *
24 Z   PZE                    *                      *
                          24  *                      *
                              ************************


R 4.633+3.450
```

Figure 5.16 - Parallel Latest Reference Search Paths

```
r run5 000000
W 1840.3
EXECUTION.
                                   1
                              ************************
                           0  *                      *(...  L3
00 A1  CLA     A              *     AC=0             *
01     STO     B              *     B=0              *      0 AC=0
02     TZE     A2             *                      *
                           2  *                      *)...  3
                              ************************
                                          .
                                          .
                                          .
                                   2      .
                              ************************
                           3  *                      *
03     CLA     X              *     AC=1             *
04     STO     Y              *     Y=1              *      3 AC=1
                           4  *                      *
                              ************************
                                          .
                                          .
                                          .
                                   3      .
                              ************************
                           5  *                      *(...  1
05 A2  CLA     B              *     AC=0             *      1 B=0
06     STO     C              *     C=0              *      5 AC=0
07     TRA     A1             *                      *
                           7  *                      *)...  L1
                              ************************


                                   4
                              ************************
                          10  *                      *
10 A   PZE     0              *                      *
11 B   BSS     1              *                      *
12 C   BSS     1              *                      *
13 X   OCT     1              *                      *
14 Y   BSS     1              *                      *
                          14  *                      *
                              ************************


R 4.600+3.083
```

search discloses two paths from location 5 to the active reference to B at location 1. The first path is from Block 3 through Block 2 to Block 1; the second path is from Block 3 directly to Block 1. The cross reference expression at location 5 states that B = 0. Thus, the functional expression for location 5 is AC = 0.

5.3.4 Multiple Latest Reference Search Paths

Figure 5.17 shows a program which contains a passive reference with multiple latest references. At location 5 there is a passive reference to X. The cross reference expressions show two latest reference values. The first is X = 1 generated by location 1 in Block 1; the second is X = 2 generated by location 4 in Block 2. Because there are two latest expressions for X at location 5, the symbol X is used in the functional expression, AC = X.

Figure 5.17 - Multiple Latest Reference Search Paths

```
r run5 000000
W 1847.4
EXECUTION.
                                   1
                              ************************
                           0  *                      *(...  L3
00 A1  CLA     A              *     AC=1             *
01     STO     X              *     X=1              *      0 AC=1
02     TZE     A2             *                      *
                           2  *                      *)...  3
                              ************************
                                          .
                                          .
                                          .
                                   2      .
                              ************************
                           3  *                      *
03     CLA     B              *     AC=2             *
04     STO     X              *     X=2              *      3 AC=2
                           4  *                      *
                              ************************
                                          .
                                          .
                                          .
                                   3      .
                              ************************
                           5  *                      *(...  1
05 A2  CLA     X              *     AC=X             *      1 X=1
                              *                      *      4 X=2
06     STO     Y              *     Y=X              *      5 AC=X
07     TRA     A1             *                      *
                           7  *                      *)...  L1
                              ************************


                                   4
                              ************************
                          10  *                      *
10 A   OCT     1              *                      *
11 B   OCT     2              *                      *
12 X   BSS     1              *                      *
13 Y   BSS     1              *                      *
                          13  *                      *
                              ************************

R 3.866+3.316
```

CHAPTER 6

CONCLUSIONS

In the previous chapters some of the problems and solutions of automatic program analysis were discussed. The initial problem that the analysis system faced was the cyclic interaction of control flow and data flow due to dependent instructions. This cyclic behavior suggests an iterative procedure in which current results were used to update and improve earlier approximations. The techniques and procedures of the first iteration were presented, and actual flowcharts of programs containing dependent instructions were displayed.

An analysis system should uncover what a program does and should transmit it to the user in a comprehensible form. The purpose of this chapter is to discuss the usefulness of the first iteration output and to suggest paths that can be followed in the second iteration to further improve the utility of these results.

6.1 THE USEFULNESS OF THE FIRST ITERATION OUTPUT

When a programmer begins to layout a program, he has a specific job or function he wishes the machine to perform. For example, he might wish to write a subroutine which calculates sine x. The programmer knows that he must develop an algorithm for calculating sine x and then convert his algorithm into machine code.

First, the programmer remembers from past experience that there is an infinite series expansion for sine x of the form:

$$\text{sine } x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots.$$

Second, the programmer knows that he must truncate the infinite series after the n-th term because his machine has limited speed and accuracy. Therefore, he develops an approximate function of the form:

$$\text{sine } x = \sum_{i=1}^{n} (-1)^i \frac{x^{2i+1}}{(2i+1)!}$$

Third, the programmer might now decide to transform his truncated series into a rational approximation or to reduce the series length by applying Chabyshev economization.

Fourth, the programmer minimizes the number of instructions and execution cycles by deriving an expression which can be imbedded in a program loop. If the third step was omitted, the expression might be of the form:

$$\text{SUM}_i = \text{SUM}_{i-1} + (-1)^i \frac{x^2}{i(i-1)} \text{SUM}_{i-1}$$

Fifth, the programmer codes his algorithms using his own personal coding conventions and programming tricks.

When a program analysis system is applied to the final program it should reverse the programming process and uncover _what_ the program does. Because there are still unprogrammed subroutines in the functional expression generation program of the first iteration, the output flowcharts for the above example cannot be shown. If all functional expression generation subroutines were available, the first iteration should output expressions at the level of the fourth step shown above, i.e. _how_ a program does _what_ it does.

In general, the output results show that it is possible to automate the initial stages of analyzing self-modifying programs. Such stages involve scanning the input program, detecting connected pieces, constructing elementary functional relationships, and pointing out trouble areas. The feasibility even at this level is open to question because the four analysis phases currently total some 11,000 instructions, pseudo-operations, and macros which assemble into nearly 100,000 memory locations. The _time-shared_ execution time averages about thirty seconds for each of the short example programs shown in Chapter 5. (Because the analysis system was developed and debugged on an experimental time-shared system, the analysis program organization was dictated by the characteristics of the time-sharing monitor, not execution time or memory length. Thus, times and lengths are somewhat exaggerated.) It is hard to give an objective evaluation as to the usefulness of the first iteration output because the missing functional generation routines made it impossible to ask a large sample of programmers to use the output in their debugging or documentation tasks. It is true that the usefulness of these output results would be improved if they were refined by a second iteration.

6.2 THE PROBLEMS OF THE SECOND ITERATION

Throughout the first iteration, many approximations were made in order to bootstrap through the control flow - data flow interaction cycle. The second iteration must check those approximations and update them if necessary. The purpose of this section is to point out and describe promising areas of further research which should improve the results of the first iteration.

Probably the first area which should be explored is the utilization of the functional expressions generated at the end of the first iteration. This would involve the development of a functional expression simplification and manipulation subroutine similar to the work being done with the LISP programming language. Such a subroutine would be used to remove the superfluous Iverson Notation symbols introduced by the many program procedural and bookkeeping operations, e.g., A/A/1 = A/1 = 1.

A second promising area is the utilization of the input data of the program being analyzed. This would require the development of a descriptive language which would convey the meaning and scope of the input data. Such additional information could be used to reduce the almost limitless possible program outcomes.

A third promising area is the development of a second iteration which would interact on-line with a human analyzer. The first iteration would handle the routine analysis functions and tell the second iteration where help was needed. The second iteration would display its current results and ask for help. After the human being decided how the situation should be handled, the second iteration would use the new directions to update its current analysis results.

APPENDIX ONE

FLOWCHARTS OF THE ANALYSIS PROGRAM

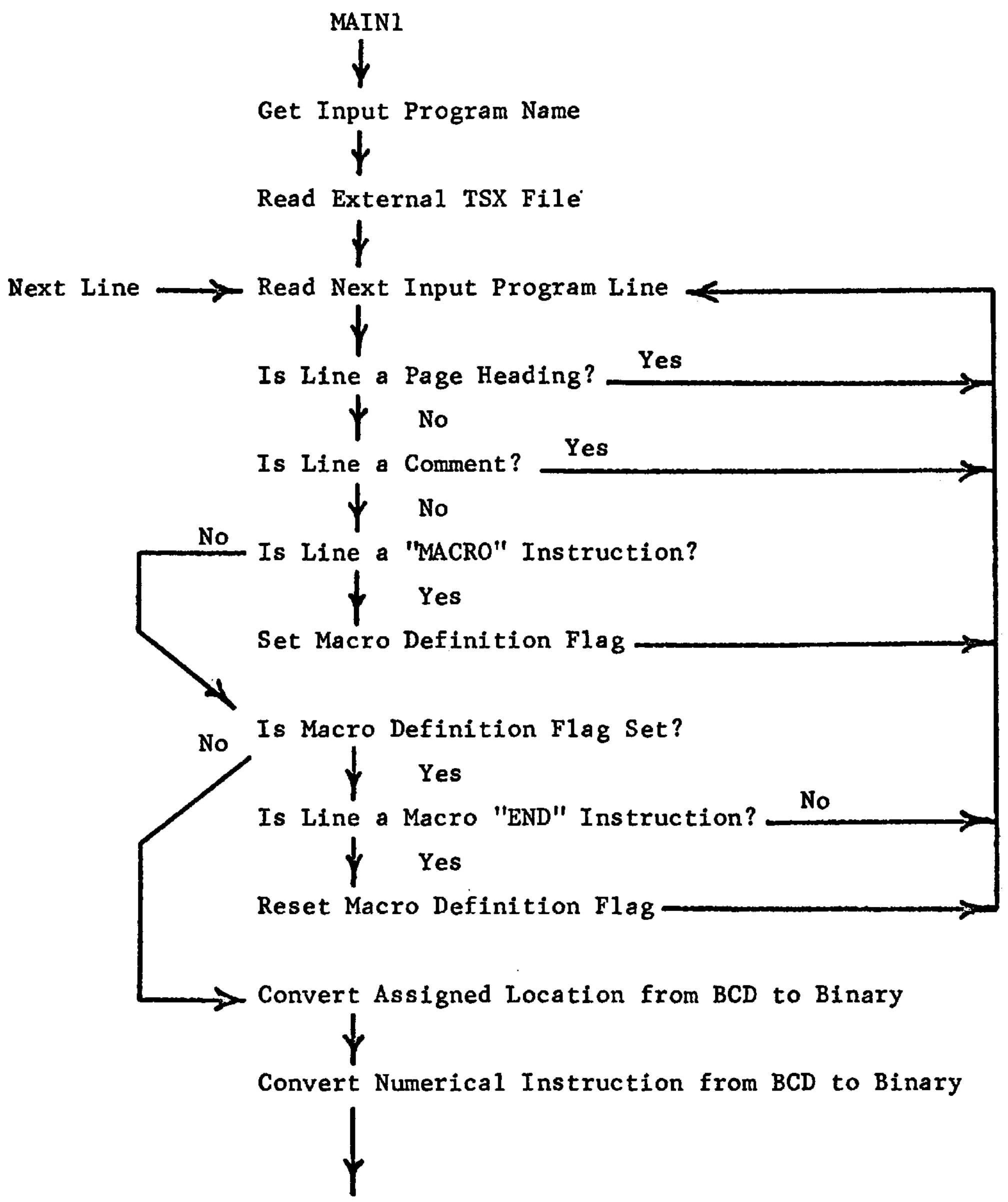
The purpose of this appendix is to present the flowcharts of the analysis program. The presentation is divided into four parts according to the analysis phases as shown in Figure 4.1. Because of the size and complexity of the analysis programs, only execution order and computation summary are shown.

PHASE ONE

MAIN1 is the main program of Phase One as shown in Figure 4.1. MAIN1 reads the input program one line at a time. Since the FAP assembler produces a variable format output tape, MAIN1 must decide what type of information is present on each line. Usually, MAIN1 will scan through the page headings, comments, and blank lines until the Transfer Vector is reached. Thereupon, the Transfer Vector entries are copied into the Transfer Vector Table. When an instruction is found, control is transferred to OPCODE for operation code identification. After OPCODE has identified the instruction and picked up its code word, RECODE recodes the instruction line into various lists as a function of the code word.

RECODE scans across the code word bit by bit. If a bit is set or on, control is transferred to its particular subroutine. Bit 1 is used to find the first executable instruction. Bit 2 is used to flag an instruction which must be treated as an exception. Bit 3 signifies a type 1 transfer, i.e. one which always transfers to the location specified by its address, e.g. a TXI instruction. Bit 4 denotes a type 1 transfer which can be tagged or indirectly addressed, e.g. a TRA instruction. Bit 5 specifies a type 2 transfer, i.e. one which can transfer control to either the address location or the next sequential location, e.g. a TXH instruction. Bit 6 signifies a type 2 transfer which can be tagged or indirectly addressed, e.g. a TZE instruction. Bit 7 shows a type 3 transfer, i.e. one which can transfer control to either of the next two sequential instructions, e.g. a ZET instruction. Bit 8 denotes a type 4 transfer, i.e. one which can transfer control to any of the next three sequential instructions, e.g. a CAS instruction. Bit 9 is reserved for the TSX instruction. Bits 10 and 11 are used by the XEC and various I/Ø instructions. Bits 12 and 13 specify Storage and Data Pseudo Operations, such as BSS and OCT. Bits 14 through 19 are reserved for the various referencing instructions. The Refer type transmits information from one location to another, e.g. a CLA instruction. The Use type uses the contents of one location to transform the contents of another location, e.g. the ORA instruction. The Test type tests the contents of various locations in order to make a transfer decision, e.g. the TZE instruction. The Set type sets the contents of a location to a known value, e.g. the STZ instruction. A Shift instruction shifts the bits of some register, e.g. the ALS instruction. An Arithmetic type performs numerical operations, e.g. the ADD instruction. Bits 27 to 36 contain a compact Short Code used to recode the instruction's operation code. The Short Codes are numbered consecutively and lend themselves to table lookups.

Figure A1.1 - MAIN1



MAIN1
Get Input Program Name
Read External TSX File
Next Line
Read Next Input Program Line
Is Line a Page Heading?
Yes
No
Is Line a Comment?
Yes
No
No
Is Line a "MACRO" Instruction?
Yes
Set Macro Definition Flag
Is Macro Definition Flag Set?
No
Yes
Is Line a Macro "END" Instruction?
No
Yes
Reset Macro Definition Flag
Convert Assigned Location from BCD to Binary
Convert Numerical Instruction from BCD to Binary

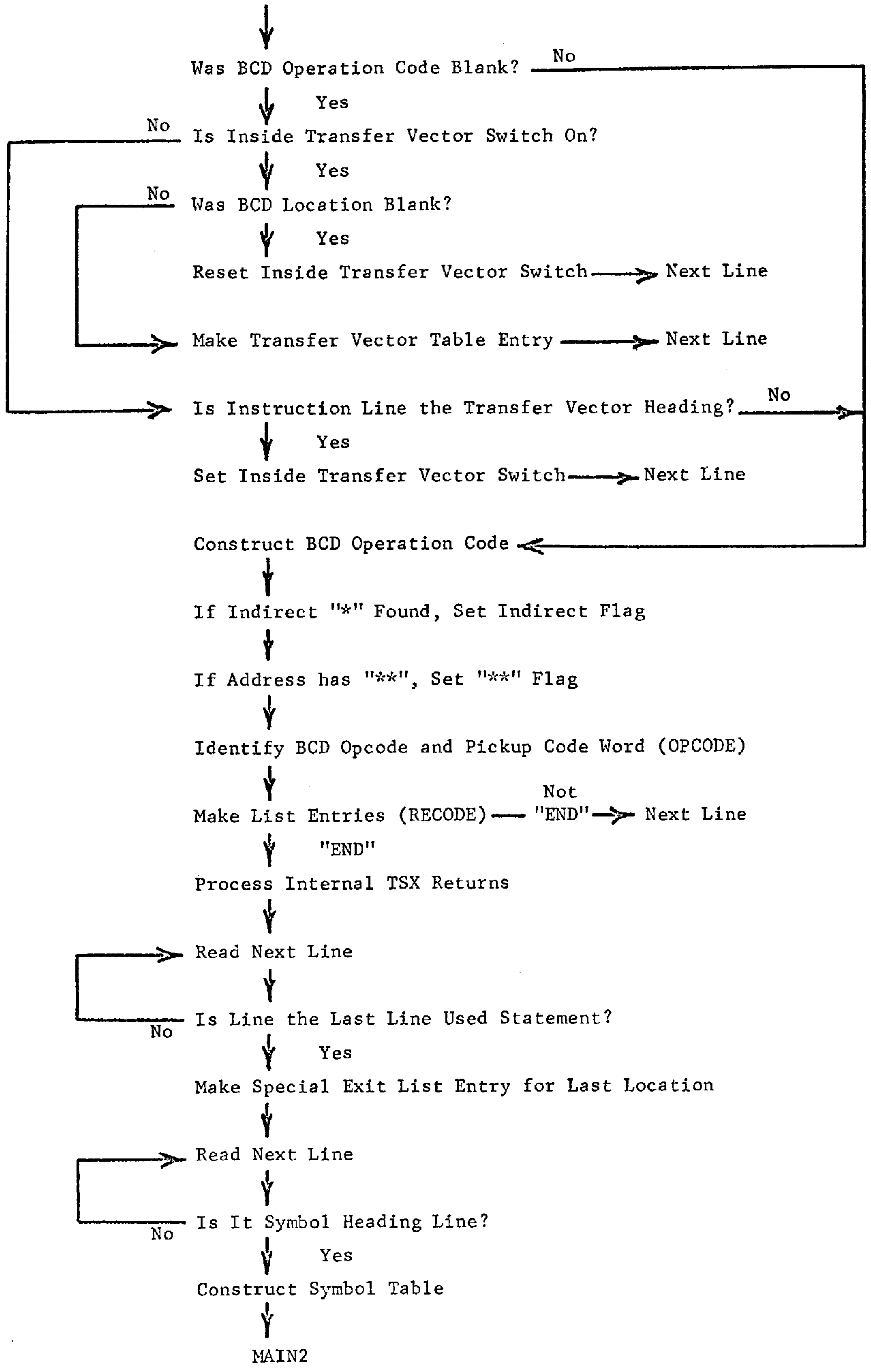
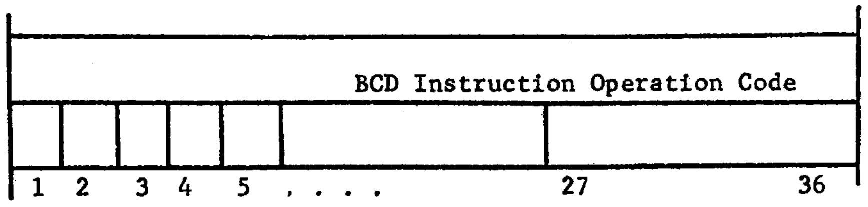

Was BCD Operation Code Blank?
No
Yes
Is Inside Transfer Vector Switch On?
No
Yes
Was BCD Location Blank?
No
Yes
Reset Inside Transfer Vector Switch
Next Line
Make Transfer Vector Table Entry
Next Line
Is Instruction Line the Transfer Vector Heading?
No
Yes
Set Inside Transfer Vector Switch
Next Line
Construct BCD Operation Code
If Indirect "*" Found, Set Indirect Flag
If Address has "**", Set "**" Flag
Identify BCD Opcode and Pickup Code Word (OPCODE)
Make List Entries (RECODE)
Not "END"
Next Line
"END"
Process Internal TSX Returns
Read Next Line
Is Line the Last Line Used Statement?
No
Yes
Make Special Exit List Entry for Last Location
Read Next Line
Is It Symbol Heading Line?
No
Yes
Construct Symbol Table
MAIN2

Figure A1.2 - OPCODE

$$\begin{tikzcd}
\text{OPCODE} \arrow[d] & \\
\text{Find Matching BCD Operation Code Entry} \arrow[d] & \\
\text{Pickup Code Word Entry} \arrow[d] & \\
\text{Transmit Code Word to RECODE} \arrow[r] & \text{MAIN1}
\end{tikzcd}$$

The OPCODE Table Entry:



Bit 1 - Executable Instruction

Bit 2 - Exception

Bit 3 - Type 1 Transfer

Bit 4 - Type 1 Tag Transfer

Bit 5 - Type 2 Transfer

Bit 6 - Type 2 Tag Transfer

Bit 7 - Type 3 Transfer

Bit 8 - Type 4 Transfer

Bit 9 - TSX Transfer

Bit 10 - XEC Instruction

Bit 11 - I/Ø Instruction

Bit 12 - Storage Pseudo Operation

Bit 13 - Data Pseudo Operation

Bit 14 - Refer Type Reference

Bit 15 - Use Type Reference

Bit 16 - Test Type Reference

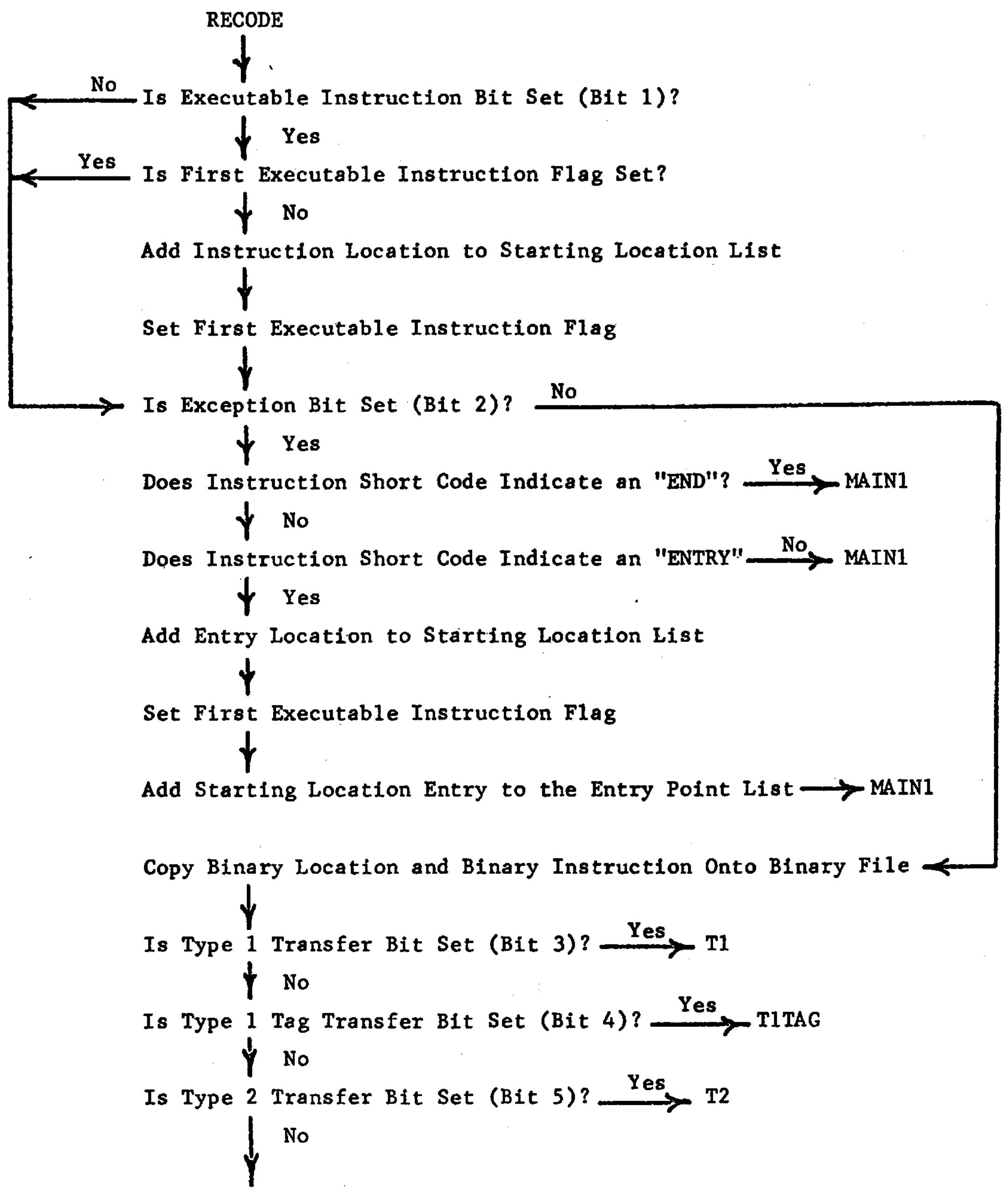
Bit 17 - Set Type Reference

Bit 18 - Shift Type Reference

Bit 19 - Arithmetic Type Reference

Bits 27 to 36 - A Compact Numerical Instruction Code used to recode the Operation Code for later table lookup identifications

Figure A1.3 - RECODE


RECODE
Is Executable Instruction Bit Set (Bit 1)?
No
Yes
Is First Executable Instruction Flag Set?
Yes
No
Add Instruction Location to Starting Location List
Set First Executable Instruction Flag
Is Exception Bit Set (Bit 2)?
No
Yes
Does Instruction Short Code Indicate an "END"?
Yes
MAIN1
No
Does Instruction Short Code Indicate an "ENTRY"
No
MAIN1
Yes
Add Entry Location to Starting Location List
Set First Executable Instruction Flag
Add Starting Location Entry to the Entry Point List
MAIN1
Copy Binary Location and Binary Instruction Onto Binary File
Is Type 1 Transfer Bit Set (Bit 3)?
Yes
T1
No
Is Type 1 Tag Transfer Bit Set (Bit 4)?
Yes
T1TAG
No
Is Type 2 Transfer Bit Set (Bit 5)?
Yes
T2
No

$$\begin{tikzcd}[row sep=small]
{} \arrow[d] & \\
\text{Is Type 2 Tag Transfer Bit Set (Bit 6)?} \arrow[r, "\text{Yes}"] \arrow[d, "\text{No}"] & \text{T2TAG} \\
\text{Is Type 3 Transfer Bit Set (Bit 7)?} \arrow[r, "\text{Yes}"] \arrow[d, "\text{No}"] & \text{T3} \\
\text{Is Type 4 Transfer Bit Set (Bit 8)?} \arrow[r, "\text{Yes}"] \arrow[d, "\text{No}"] & \text{T4} \\
\text{Is TSX Transfer Bit Set (Bit 9)?} \arrow[r, "\text{Yes}"] \arrow[d, "\text{No}"] & \text{TSXTRN} \\
\text{Is XEC Bit Set (Bit 10)?} \arrow[r, "\text{Yes}"] \arrow[d, "\text{No}"] & \text{XEC (Not Programmed)} \\
\text{Is I/Ø Bit Set (Bit 11)?} \arrow[r, "\text{Yes}"] \arrow[d, "\text{No}"] & \text{I/Ø (Not Programmed)} \\
\text{Is Storage Bit Set (Bit 12)?} \arrow[r, "\text{Yes}"] \arrow[d, "\text{No}"] & \text{STORAG} \\
\text{Is Data Bit Set (Bit 13)?} \arrow[r, "\text{Yes}"] \arrow[d, "\text{No}"] & \text{DATGEN} \\
\text{Is Refer Type Reference Bit Set (Bit 14)?} \arrow[r, "\text{Yes}"] \arrow[d, "\text{No}"] & \text{REFER} \\
\text{Is Use Type Reference Bit Set (Bit 15)?} \arrow[r, "\text{Yes}"] \arrow[d, "\text{No}"] & \text{USE} \\
\text{Is Test Type Reference Bit Set (Bit 16)?} \arrow[r, "\text{Yes}"] \arrow[d, "\text{No}"] & \text{TEST} \\
\text{Is Set Type Reference Bit Set (Bit 17)?} \arrow[r, "\text{Yes}"] \arrow[d, "\text{No}"] & \text{SET} \\
\text{Is Shift Type Reference Bit Set (Bit 18)?} \arrow[r, "\text{Yes}"] \arrow[d, "\text{No}"] & \text{SHIFT} \\
\text{Is Arithmetic Type Reference Bit Set (Bit 19)?} \arrow[r, "\text{Yes}"] \arrow[d, "\text{No}"] & \text{ARITH} \\
\text{MAIN1} &
\end{tikzcd}$$

Figure A1.4 - T1

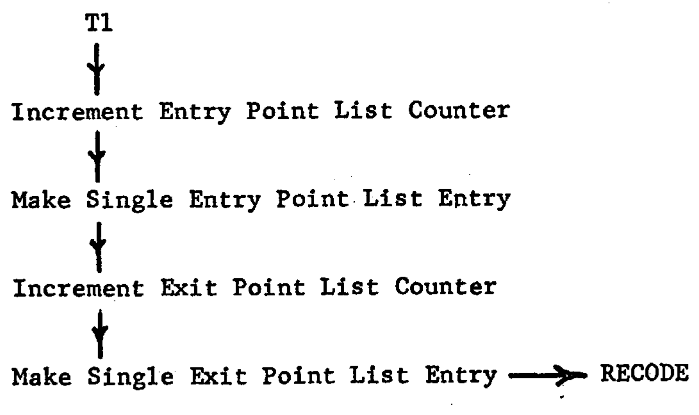

T1
Increment Entry Point List Counter
Make Single Entry Point List Entry
Increment Exit Point List Counter
Make Single Exit Point List Entry
RECODE

Figure A1.5 - T1TAG

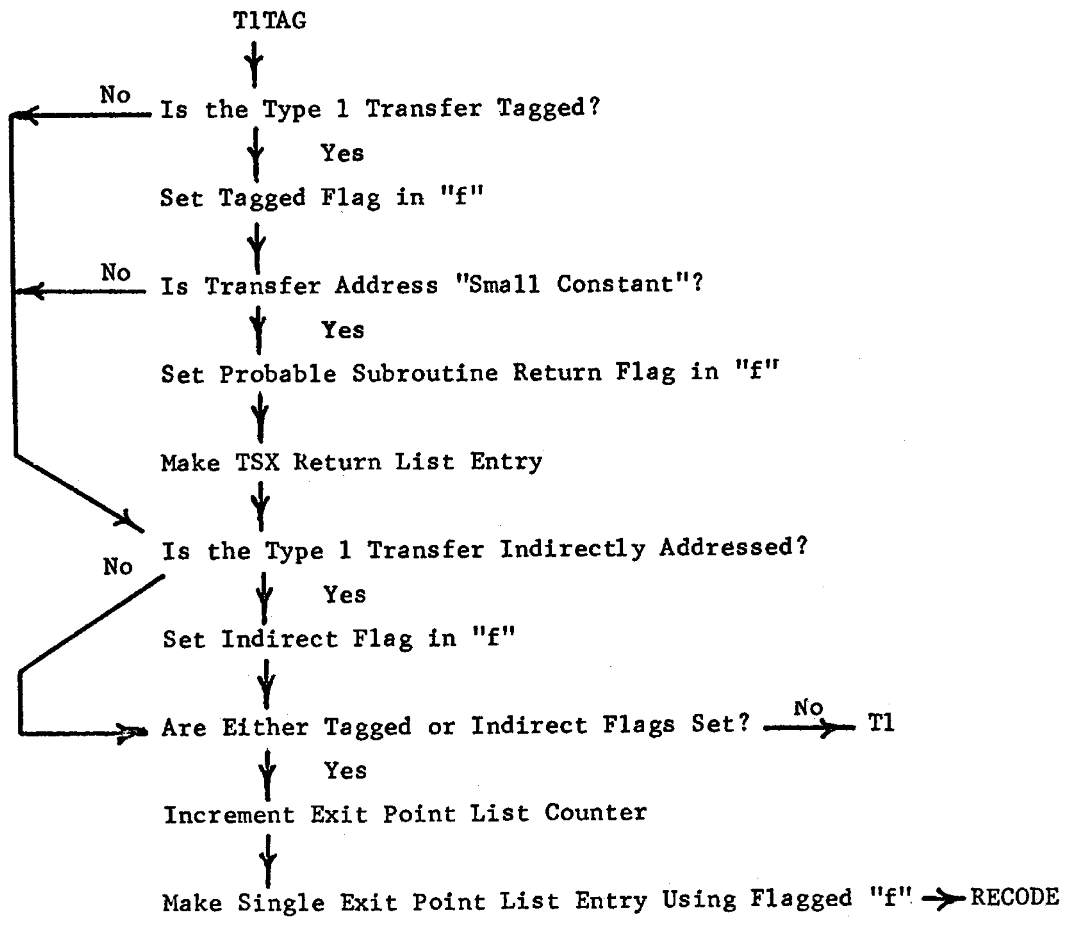

T1TAG
Is the Type 1 Transfer Tagged?
No
Yes
Set Tagged Flag in "f"
Is Transfer Address "Small Constant"?
No
Yes
Set Probable Subroutine Return Flag in "f"
Make TSX Return List Entry
Is the Type 1 Transfer Indirectly Addressed?
No
Yes
Set Indirect Flag in "f"
Are Either Tagged or Indirect Flags Set?
No
T1
Yes
Increment Exit Point List Counter
Make Single Exit Point List Entry Using Flagged "f"
RECODE

Figure A1.6 - T2

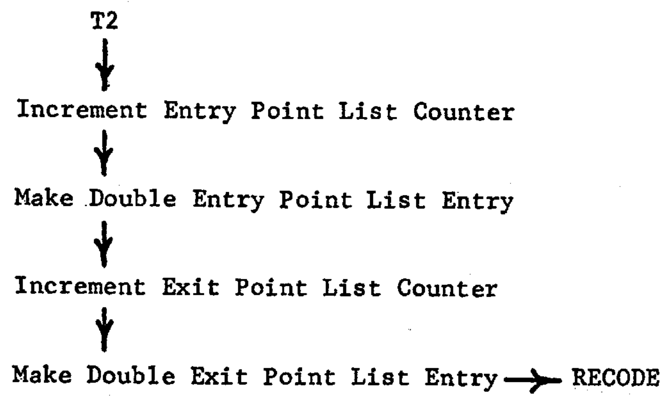

T2
Increment Entry Point List Counter
Make Double Entry Point List Entry
Increment Exit Point List Counter
Make Double Exit Point List Entry
RECODE

Figure A1.7 - T2TAG

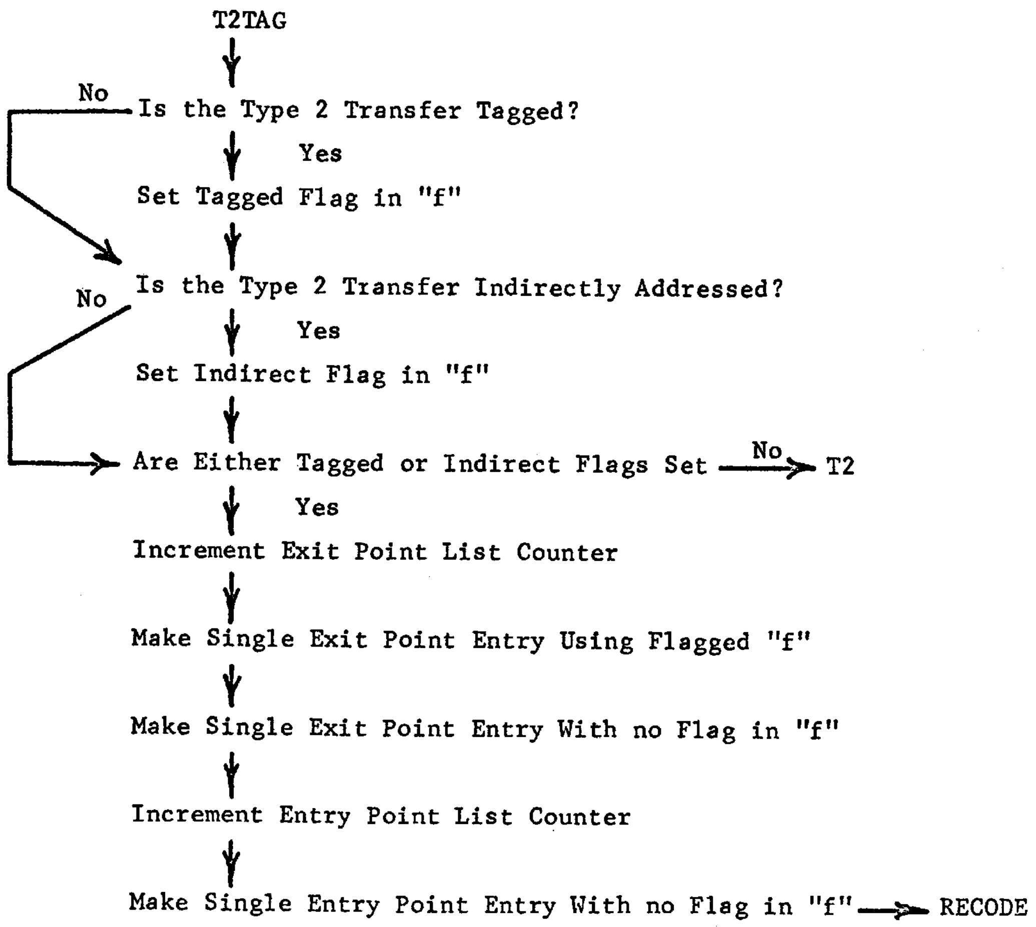

T2TAG
Is the Type 2 Transfer Tagged?
No
Yes
Set Tagged Flag in "f"
Is the Type 2 Transfer Indirectly Addressed?
No
Yes
Set Indirect Flag in "f"
Are Either Tagged or Indirect Flags Set
No
T2
Yes
Increment Exit Point List Counter
Make Single Exit Point Entry Using Flagged "f"
Make Single Exit Point Entry With no Flag in "f"
Increment Entry Point List Counter
Make Single Entry Point Entry With no Flag in "f"
RECODE

Figure A1.8 - T3

$$\begin{tikzcd}
\text{T3} \arrow[d] & \\
\text{Increment Entry Point List Counter} \arrow[d] & \\
\text{Make Two Entry Point List Entries} \arrow[d] & \\
\text{Increment Exit Point List Counter} \arrow[d] & \\
\text{Make Two Exit Point List Entries} \arrow[r] & \text{RECODE}
\end{tikzcd}$$

Figure A1.9 - T4

$$\begin{tikzcd}
\text{T4} \arrow[d] & \\
\text{Increment Entry Point List Counter} \arrow[d] & \\
\text{Make Three Entry Point List Entries} \arrow[d] & \\
\text{Increment Exit Point List Counter} \arrow[d] & \\
\text{Make Three Exit Point List Entries} \arrow[r] & \text{RECODE}
\end{tikzcd}$$

Figure A1.10 - TSXTRN

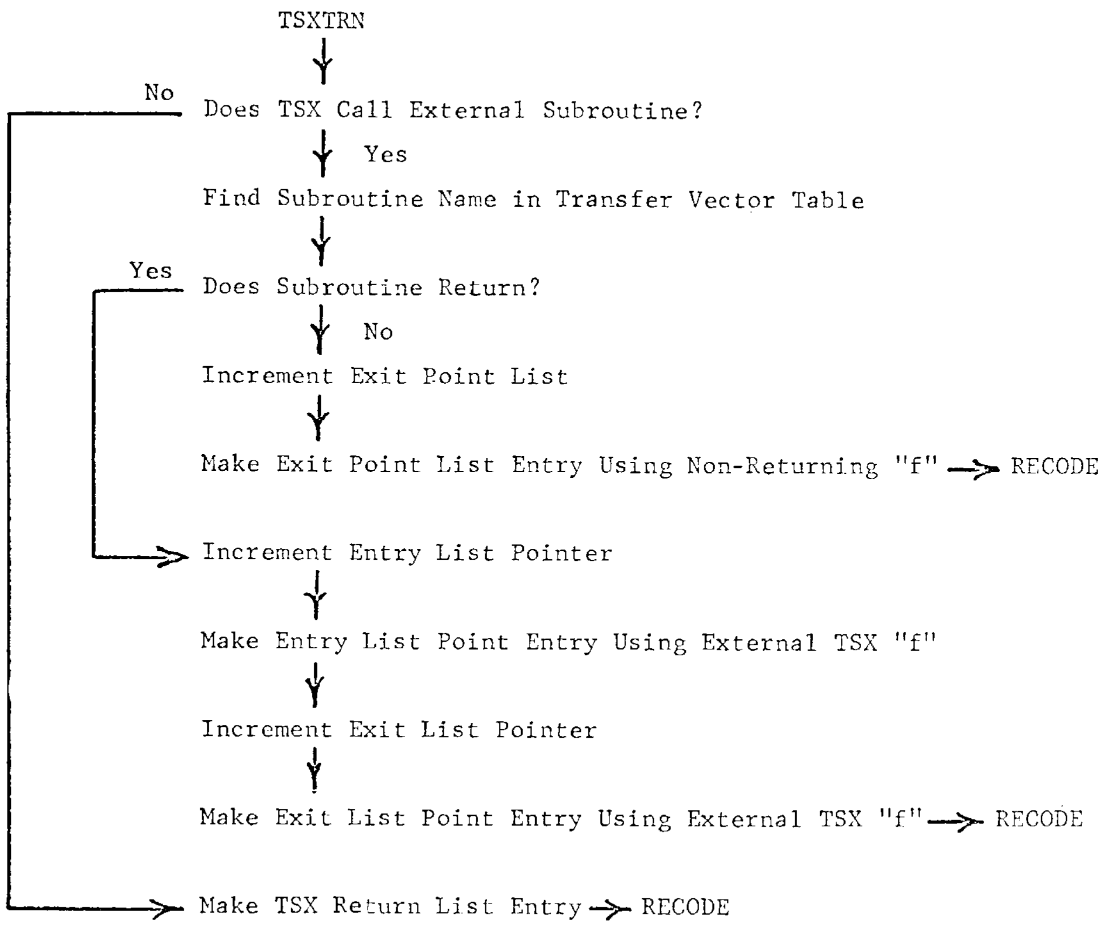

TSXTRN
No
Does TSX Call External Subroutine?
Yes
Find Subroutine Name in Transfer Vector Table
Yes
Does Subroutine Return?
No
Increment Exit Point List
Make Exit Point List Entry Using Non-Returning "f"
RECODE
Increment Entry List Pointer
Make Entry List Point Entry Using External TSX "f"
Increment Exit List Pointer
Make Exit List Point Entry Using External TSX "f"
RECODE
Make TSX Return List Entry
RECODE

Figure A1.11 - DATGEN

$$\begin{tikzcd}[column sep=small]
\text{DATGEN} \arrow[d] & \\
\text{Increment Data List Counter} \arrow[d] & \\
\text{Find Number of Locations Generated by the Data Pseudo Operation} \arrow[d] & \\
\text{Construct Data List Entry Showing First and Last Location} \arrow[d] & \\
\text{Make Data List Entry} \arrow[r] & \text{RECODE}
\end{tikzcd}$$

Figure A1.12 - STORAG

$$\begin{tikzcd}[column sep=small]
\text{STORAG} \arrow[d] & \\
\text{Increment Data List Counter} \arrow[d] & \\
\text{Find Number of Locations Reserved by the Storage Pseudo Operation} \arrow[d] & \\
\text{Construct Storage List Entry Showing First and Last Location} \arrow[d] & \\
\text{Make Storage List Entry} \arrow[r] & \text{RECODE}
\end{tikzcd}$$

Figure A1.13 - REFER

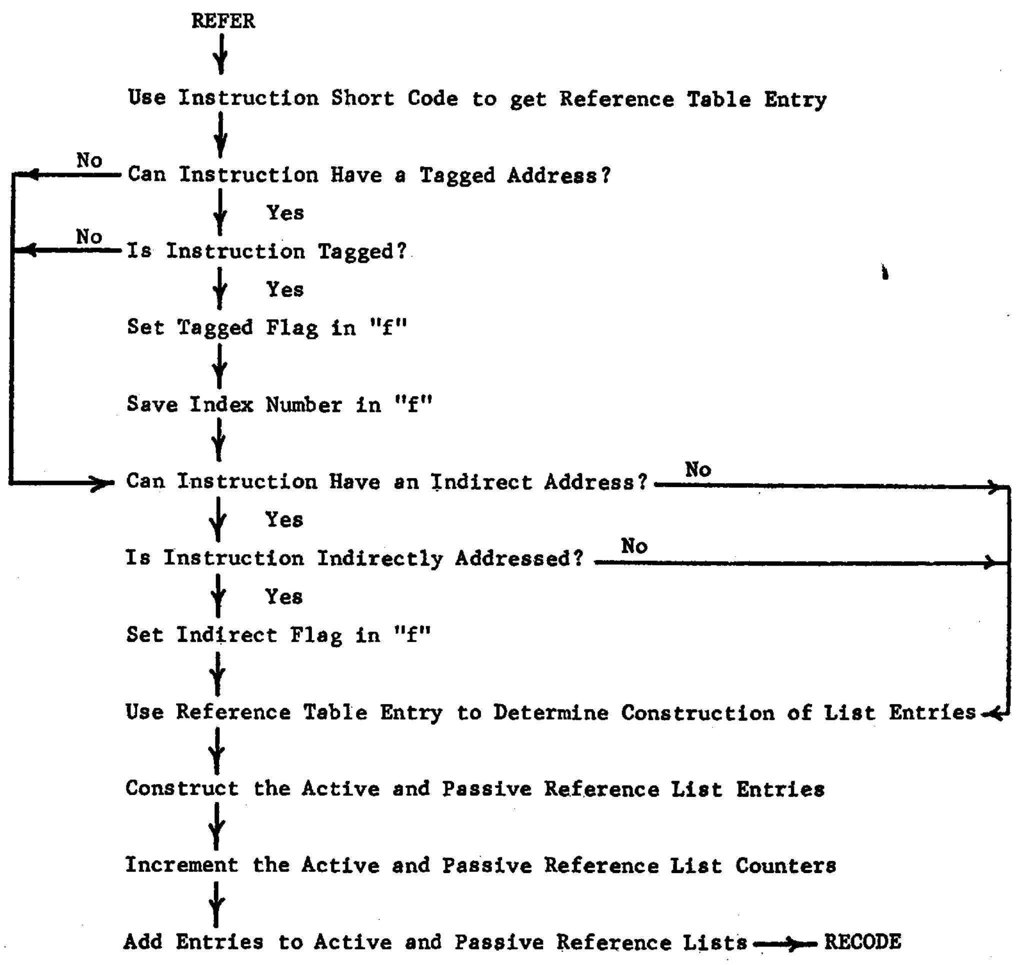

REFER
Use Instruction Short Code to get Reference Table Entry
Can Instruction Have a Tagged Address?
No
Yes
Is Instruction Tagged?
No
Yes
Set Tagged Flag in "f"
Save Index Number in "f"
Can Instruction Have an Indirect Address?
No
Yes
Is Instruction Indirectly Addressed?
No
Yes
Set Indirect Flag in "f"
Use Reference Table Entry to Determine Construction of List Entries
Construct the Active and Passive Reference List Entries
Increment the Active and Passive Reference List Counters
Add Entries to Active and Passive Reference Lists
RECODE

Figure A1.14 - USE

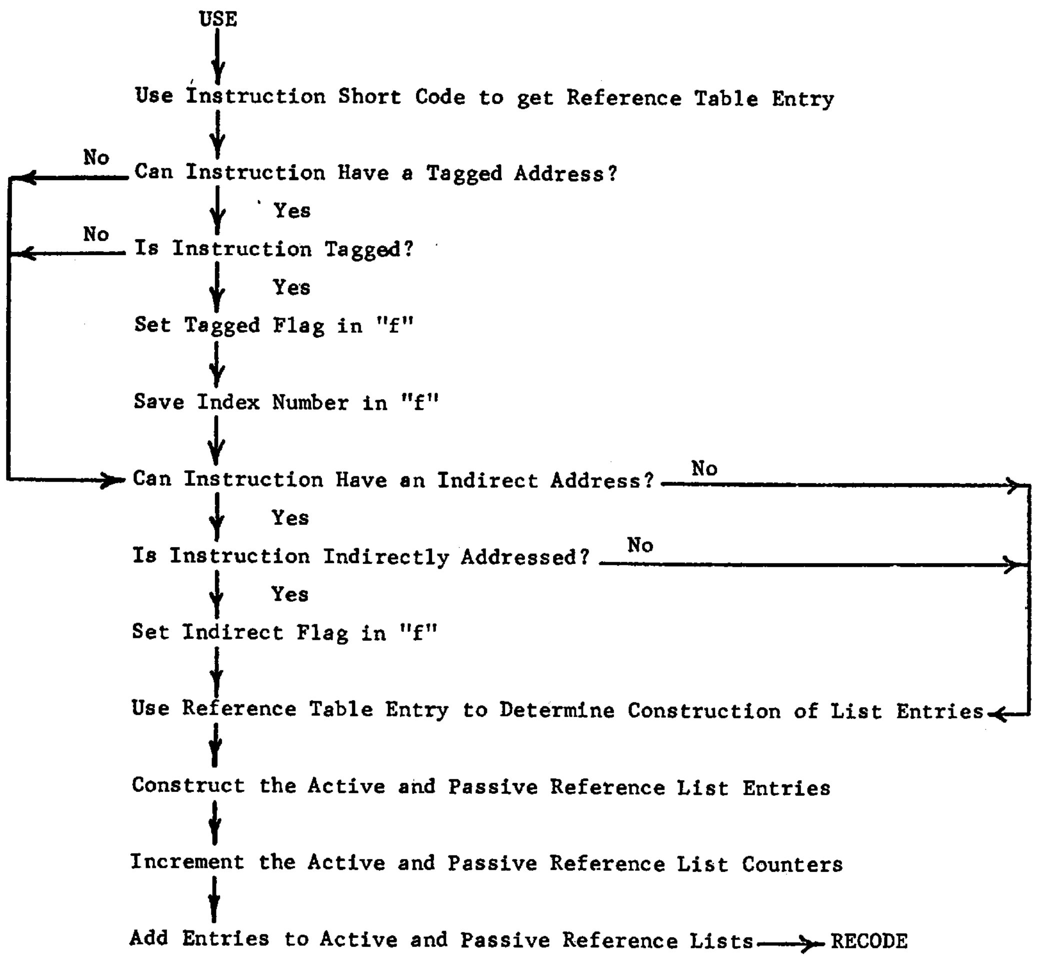

USE
Use Instruction Short Code to get Reference Table Entry
Can Instruction Have a Tagged Address?
No
Yes
Is Instruction Tagged?
No
Yes
Set Tagged Flag in "f"
Save Index Number in "f"
Can Instruction Have an Indirect Address?
No
Yes
Is Instruction Indirectly Addressed?
No
Yes
Set Indirect Flag in "f"
Use Reference Table Entry to Determine Construction of List Entries
Construct the Active and Passive Reference List Entries
Increment the Active and Passive Reference List Counters
Add Entries to Active and Passive Reference Lists
RECODE

Figure A1.15 - TEST

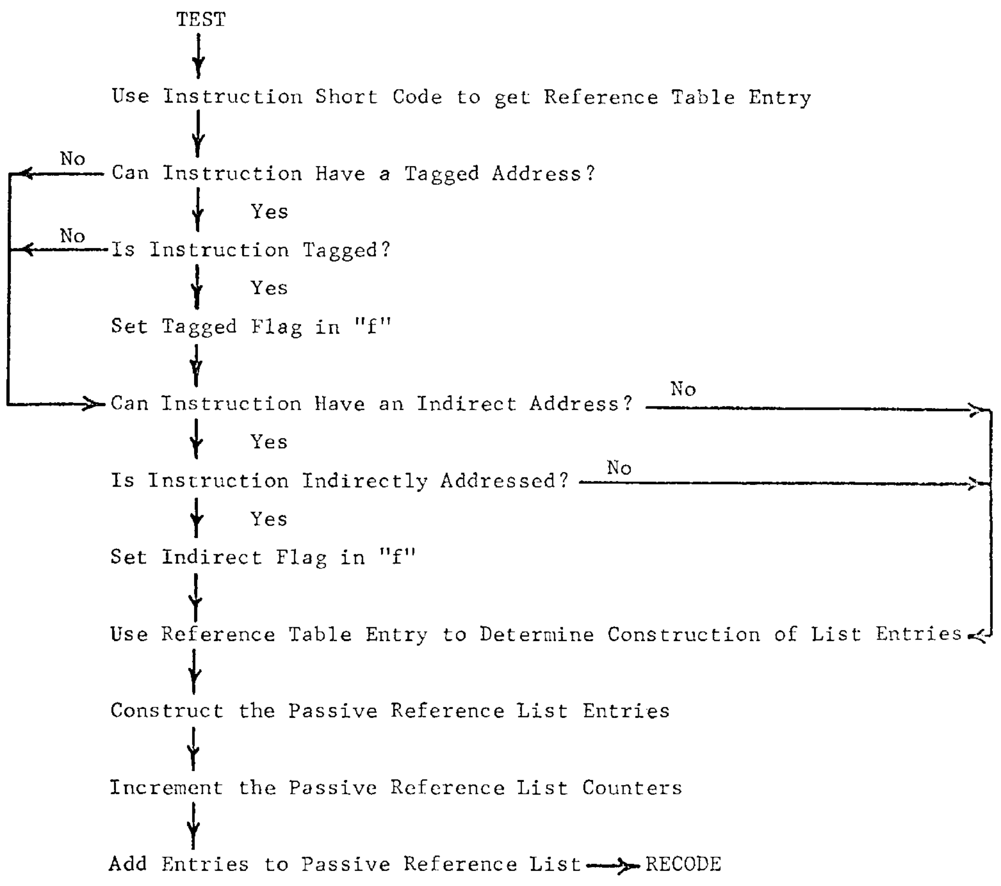

TEST
Use Instruction Short Code to get Reference Table Entry
Can Instruction Have a Tagged Address?
No
Yes
Is Instruction Tagged?
No
Yes
Set Tagged Flag in "f"
Can Instruction Have an Indirect Address?
No
Yes
Is Instruction Indirectly Addressed?
No
Yes
Set Indirect Flag in "f"
Use Reference Table Entry to Determine Construction of List Entries
Construct the Passive Reference List Entries
Increment the Passive Reference List Counters
Add Entries to Passive Reference List
RECODE

Figure A1.16 - SET

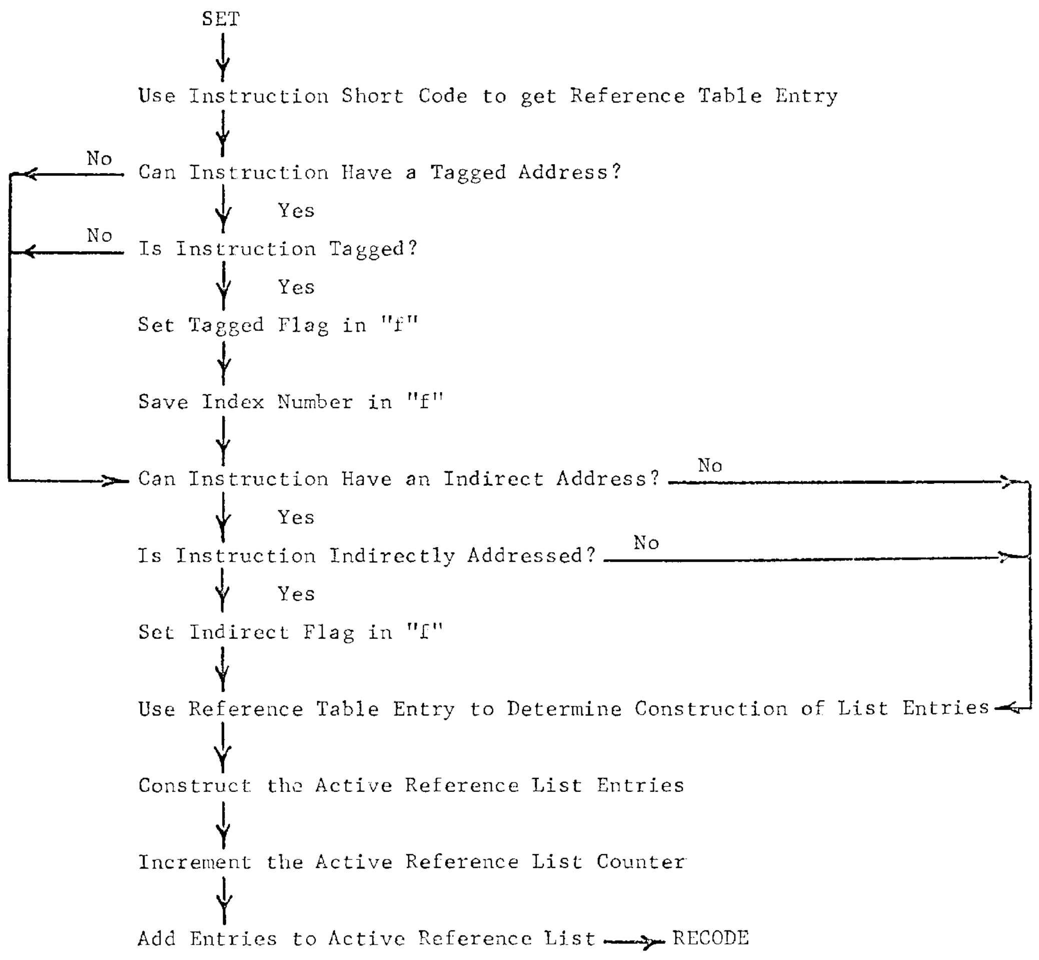

SET
Use Instruction Short Code to get Reference Table Entry
No
Can Instruction Have a Tagged Address?
Yes
No
Is Instruction Tagged?
Yes
Set Tagged Flag in "f"
Save Index Number in "f"
Can Instruction Have an Indirect Address?
No
Yes
Is Instruction Indirectly Addressed?
No
Yes
Set Indirect Flag in "f"
Use Reference Table Entry to Determine Construction of List Entries
Construct the Active Reference List Entries
Increment the Active Reference List Counter
Add Entries to Active Reference List
RECODE

Figure A1.17 - SHIFT

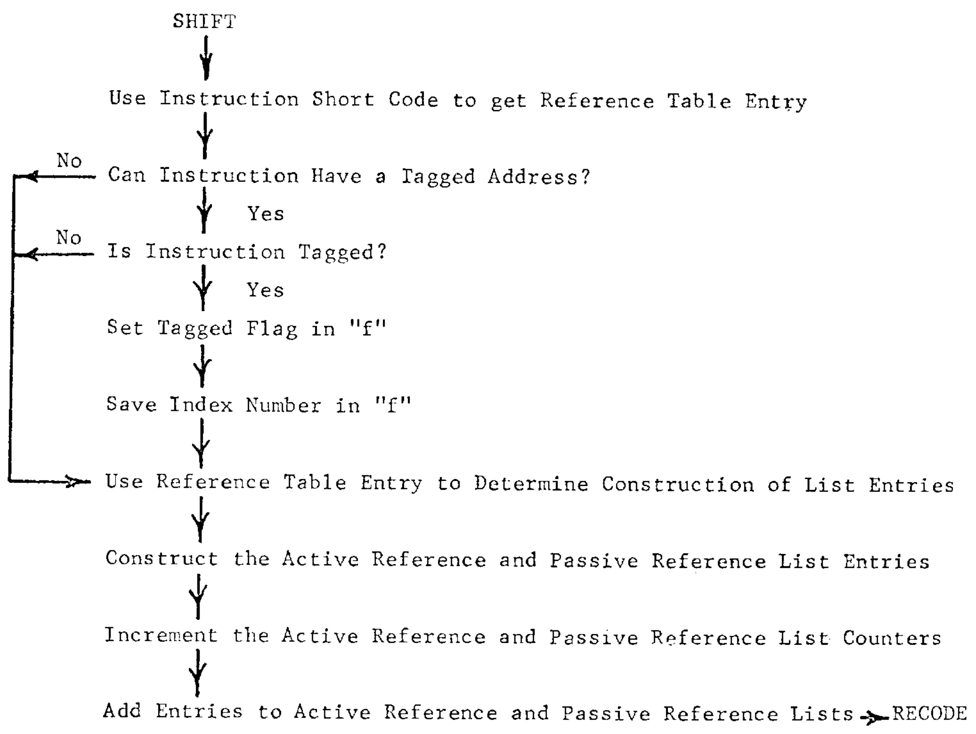
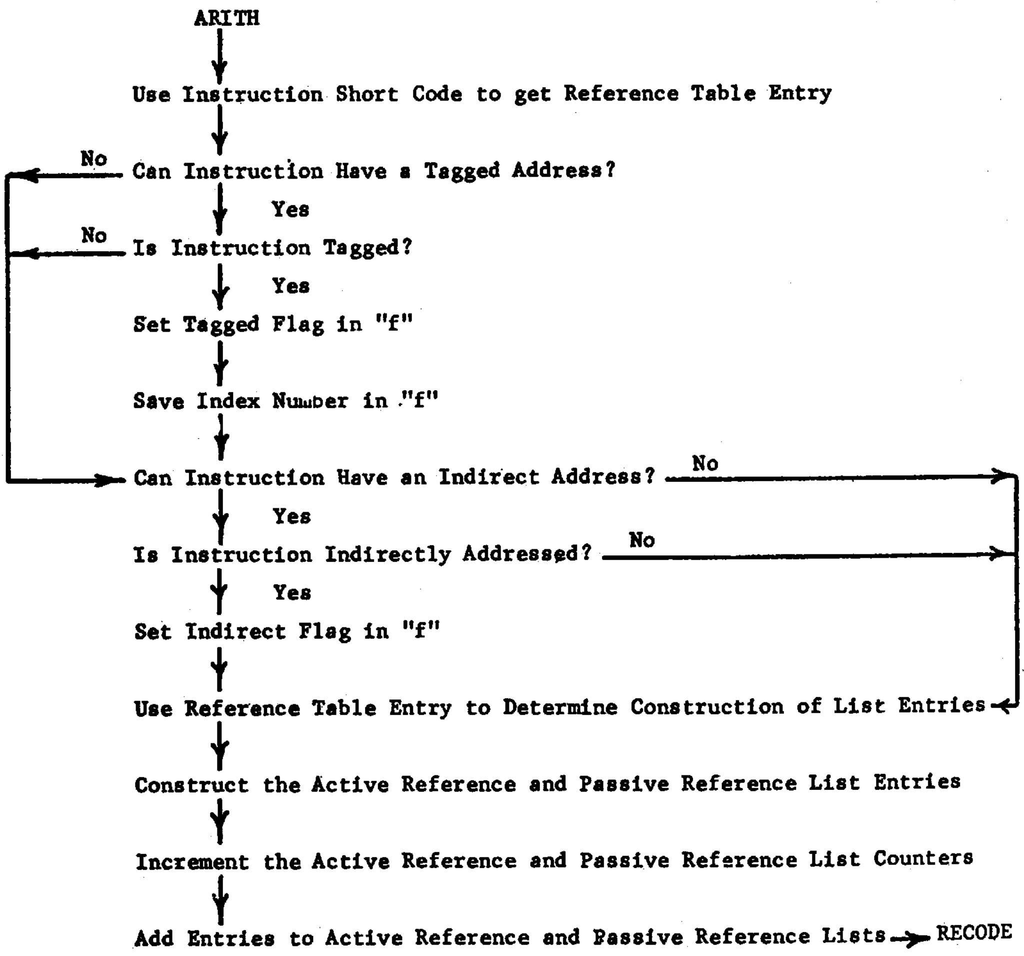

SHIFT
Use Instruction Short Code to get Reference Table Entry
No
Can Instruction Have a Tagged Address?
Yes
No
Is Instruction Tagged?
Yes
Set Tagged Flag in "f"
Save Index Number in "f"
Use Reference Table Entry to Determine Construction of List Entries
Construct the Active Reference and Passive Reference List Entries
Increment the Active Reference and Passive Reference List Counters
Add Entries to Active Reference and Passive Reference Lists
RECODE

Figure A1.18 - ARITH


ARITH
Use Instruction Short Code to get Reference Table Entry
Can Instruction Have a Tagged Address?
No
Yes
Is Instruction Tagged?
No
Yes
Set Tagged Flag in "f"
Save Index Number in "f"
Can Instruction Have an Indirect Address?
No
Yes
Is Instruction Indirectly Addressed?
No
Yes
Set Indirect Flag in "f"
Use Reference Table Entry to Determine Construction of List Entries
Construct the Active Reference and Passive Reference List Entries
Increment the Active Reference and Passive Reference List Counters
Add Entries to Active Reference and Passive Reference Lists
RECODE

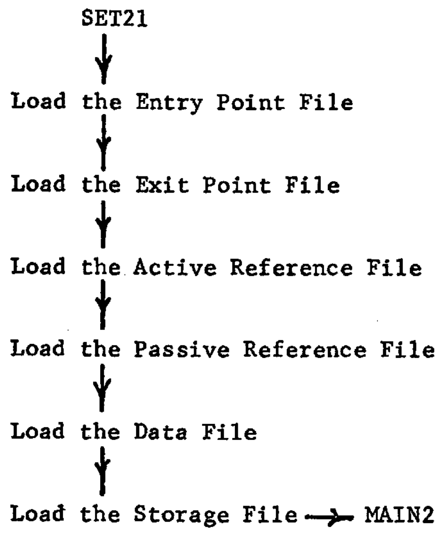
PHASE TWO

MAIN2 is the main program of Phase Two as shown in Figure 4.1. MAIN2 calls seven subroutines which perform the required Data Processing functions. Because of programming considerations, the Data Reduction function of breaking the program into blocks is performed at the end of this phase. SET21 reads the various temporary data files into memory. PART finds which portions of each cell are actively referenced. CONSAT determines which passive reference entries reference constants and which active reference entries reference results. GETCON finds the value of each constant cell by scanning the Binary File. SWITCH detects any transfer switches and corrects the Entry Point and Exit Point Lists. CHANGE identifies and flags all modified instructions. TOPSET breaks the program into blocks and constructs the Control Tables.

Figure A1.19 - MAIN2

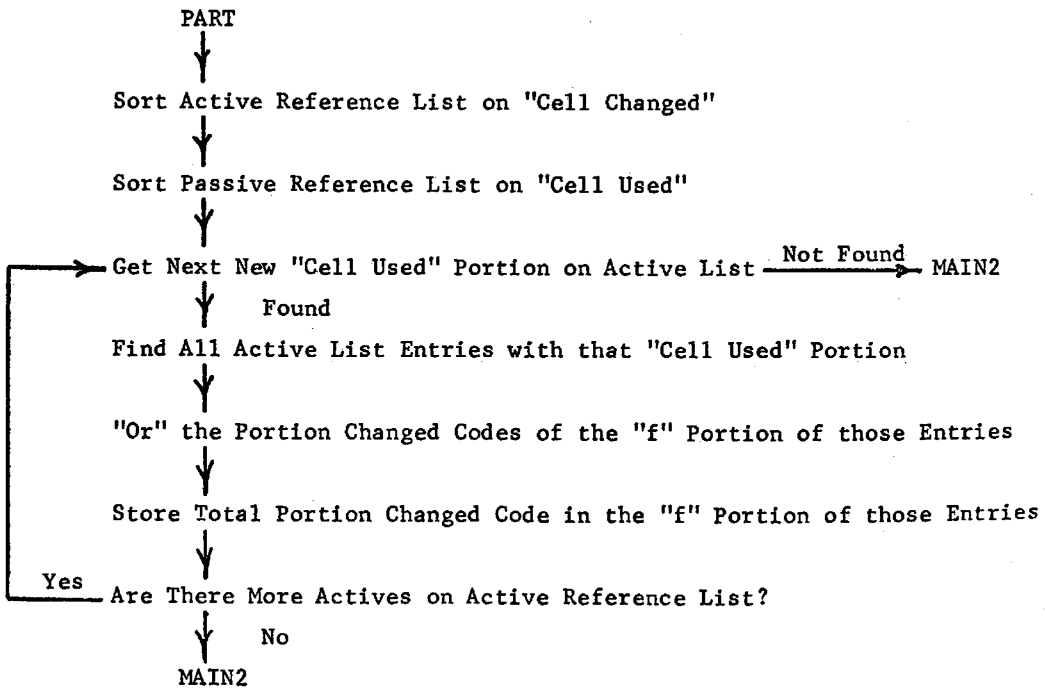
$$\begin{tikzcd}
\text{MAIN2} \arrow[d] \\
\text{Read Various Files into Memory (SET21)} \arrow[d] \\
\text{Find Total Portion Changed (PART)} \arrow[d] \\
\text{Find Constants and Results (CONSAT)} \arrow[d] \\
\text{Get the Value of the Constant Locations (GETCON)} \arrow[d] \\
\text{Find Changed Instructions (CHANGE)} \arrow[d] \\
\text{Find Transfer Switches (SWITCH)} \arrow[d] \\
\text{Find Program Topology (TOPSET)} \arrow[d] \\
\text{MAIN3}
\end{tikzcd}$$

Figure A1.20 - SET21



SET21
Load the Entry Point File
Load the Exit Point File
Load the Active Reference File
Load the Passive Reference File
Load the Data File
Load the Storage File
MAIN2

Figure A1.21 - PART



PART
Sort Active Reference List on "Cell Changed"
Sort Passive Reference List on "Cell Used"
Get Next New "Cell Used" Portion on Active List
Not Found
MAIN2
Found
Find All Active List Entries with that "Cell Used" Portion
"Or" the Portion Changed Codes of the "f" Portion of those Entries
Store Total Portion Changed Code in the "f" Portion of those Entries
Yes
Are There More Actives on Active Reference List?
No
MAIN2

Figure A1.22 - CONSAT

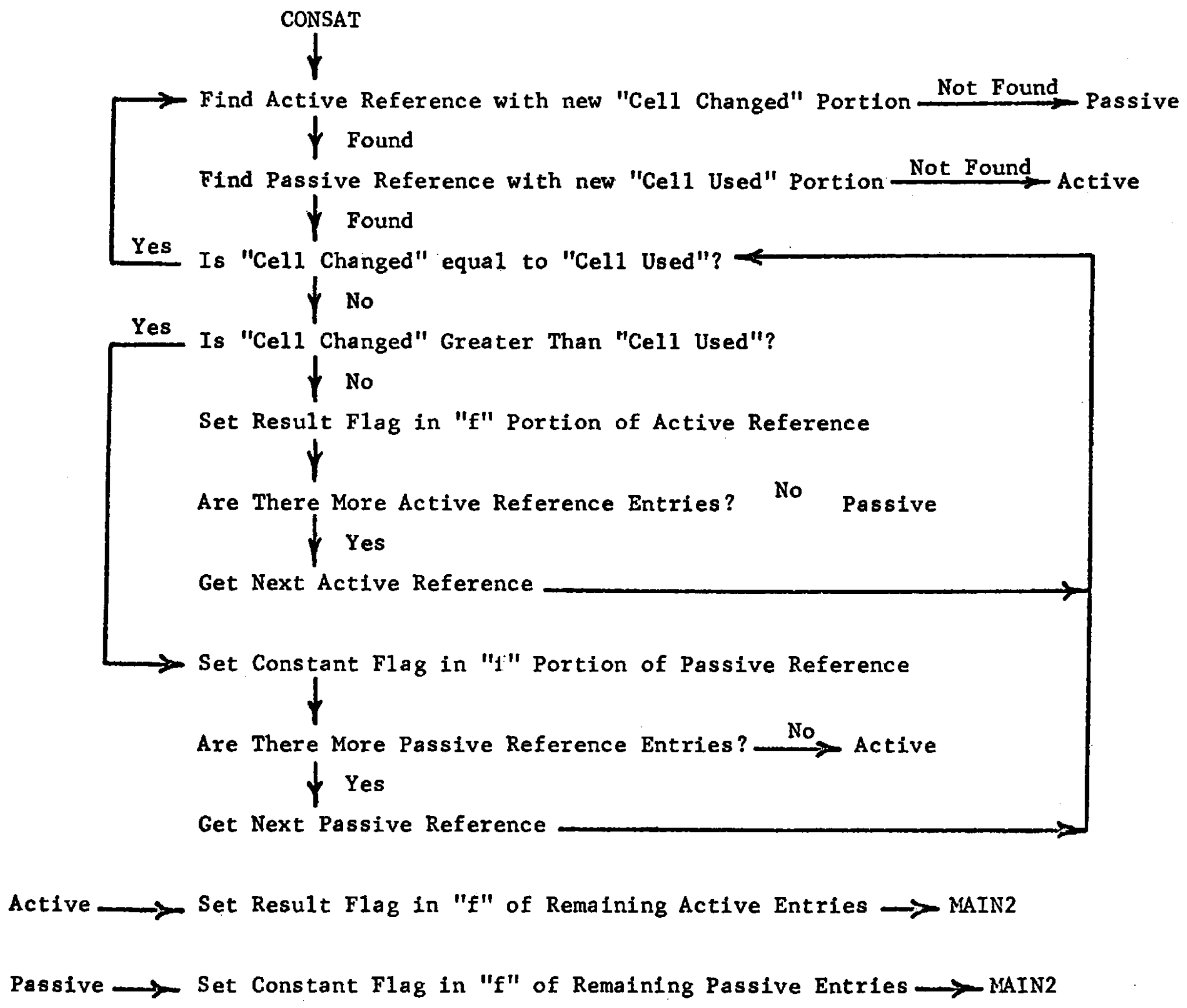

CONSAT
Find Active Reference with new "Cell Changed" Portion
Not Found
Passive
Found
Find Passive Reference with new "Cell Used" Portion
Not Found
Active
Found
Yes
Is "Cell Changed" equal to "Cell Used"?
No
Yes
Is "Cell Changed" Greater Than "Cell Used"?
No
Set Result Flag in "f" Portion of Active Reference
Are There More Active Reference Entries?
No
Passive
Yes
Get Next Active Reference
Set Constant Flag in "f" Portion of Passive Reference
Are There More Passive Reference Entries?
No
Active
Yes
Get Next Passive Reference
Active
Set Result Flag in "f" of Remaining Active Entries
MAIN2
Passive
Set Constant Flag in "f" of Remaining Passive Entries
MAIN2

Figure A1.23 - GETCON

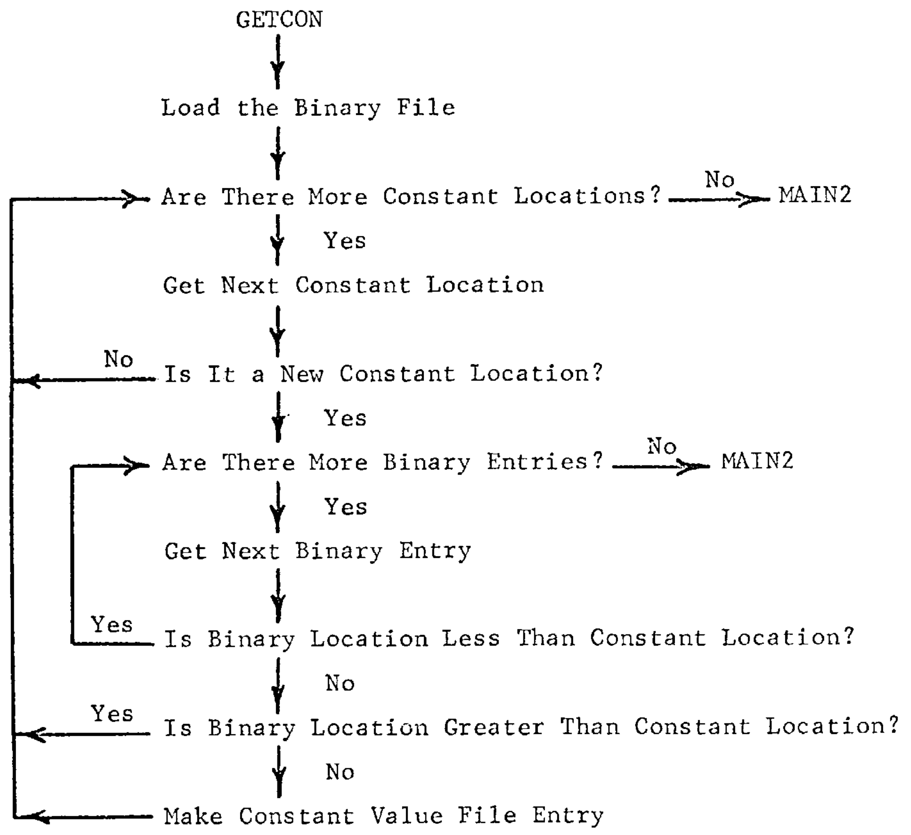
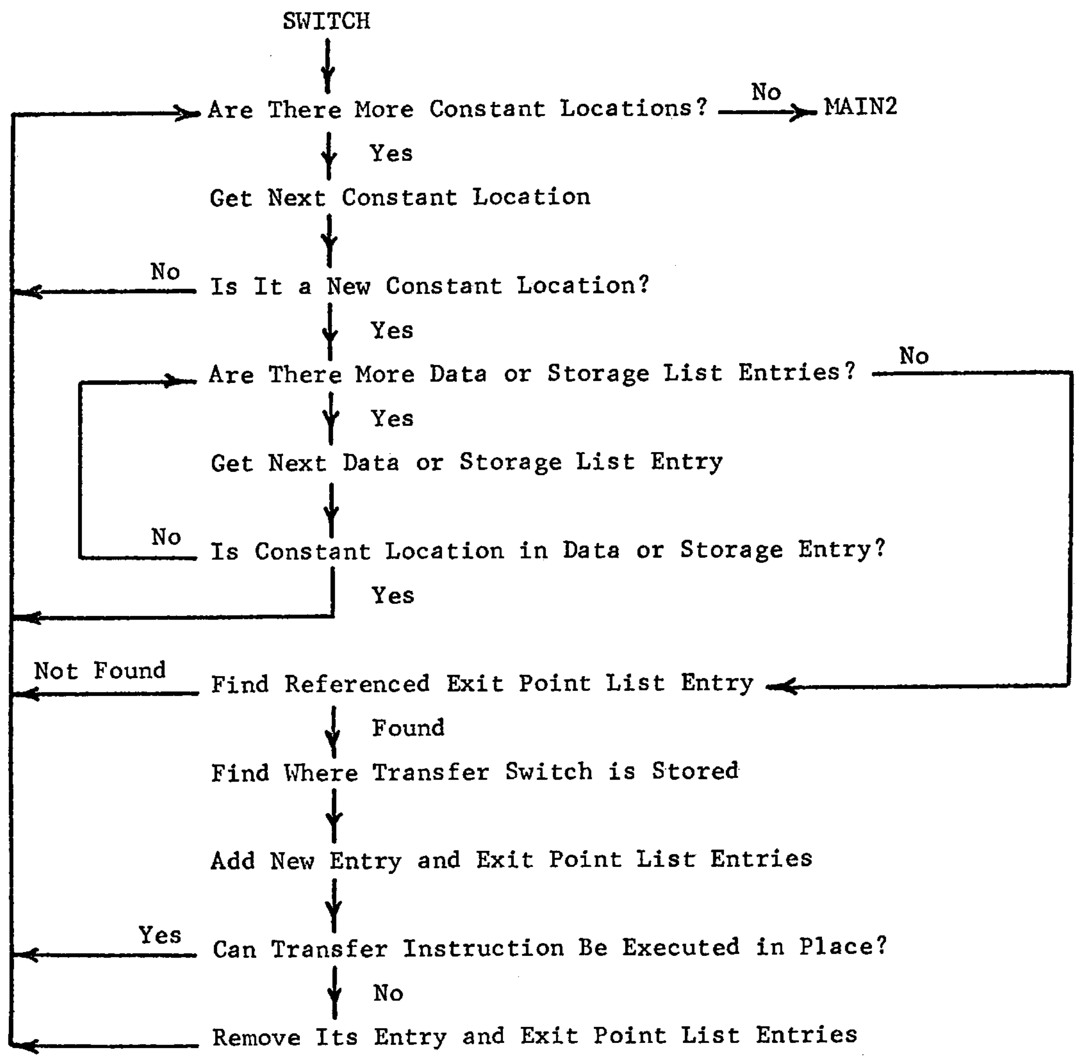

GETCON
Load the Binary File
Are There More Constant Locations?
No
MAIN2
Yes
Get Next Constant Location
No
Is It a New Constant Location?
Yes
Are There More Binary Entries?
No
MAIN2
Yes
Get Next Binary Entry
Yes
Is Binary Location Less Than Constant Location?
No
Yes
Is Binary Location Greater Than Constant Location?
No
Make Constant Value File Entry

Figure A1.24 - SWITCH



SWITCH
Are There More Constant Locations?
No
MAIN2
Yes
Get Next Constant Location
No
Is It a New Constant Location?
Yes
Are There More Data or Storage List Entries?
No
Yes
Get Next Data or Storage List Entry
No
Is Constant Location in Data or Storage Entry?
Yes
Not Found
Find Referenced Exit Point List Entry
Found
Find Where Transfer Switch is Stored
Add New Entry and Exit Point List Entries
Yes
Can Transfer Instruction Be Executed in Place?
No
Remove Its Entry and Exit Point List Entries

Figure A1.25 - CHANGE

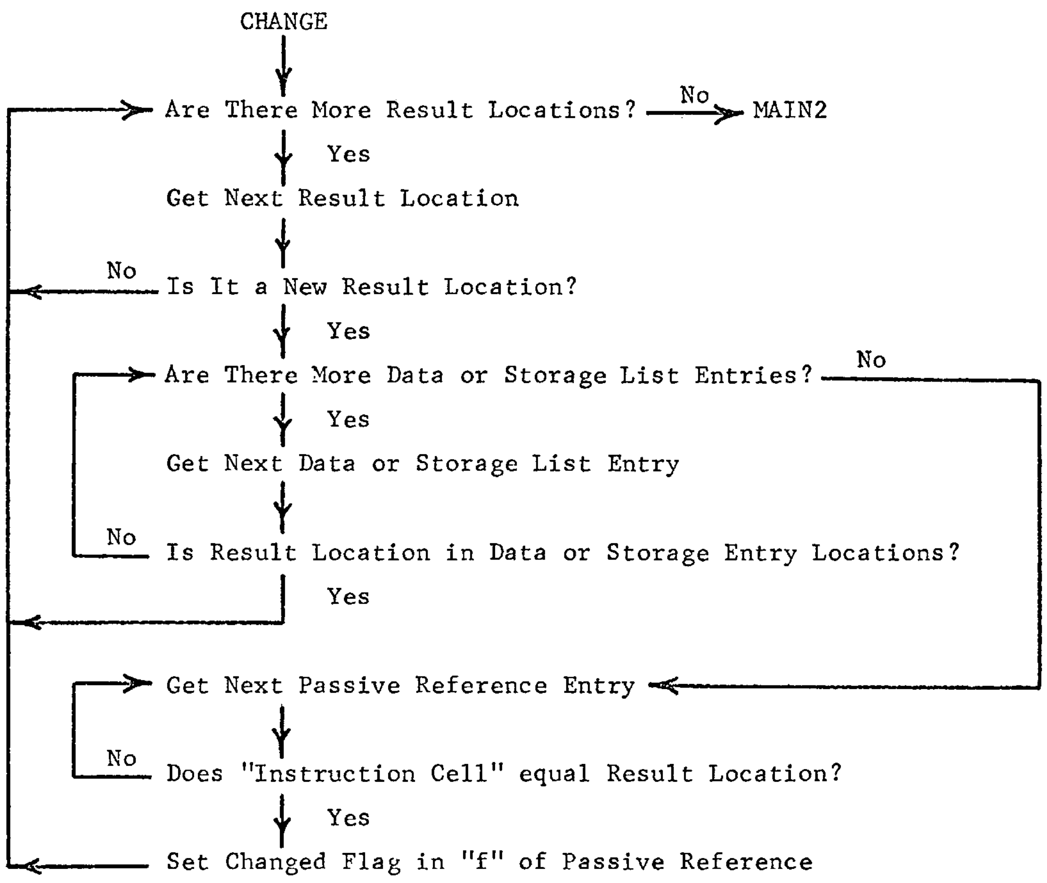
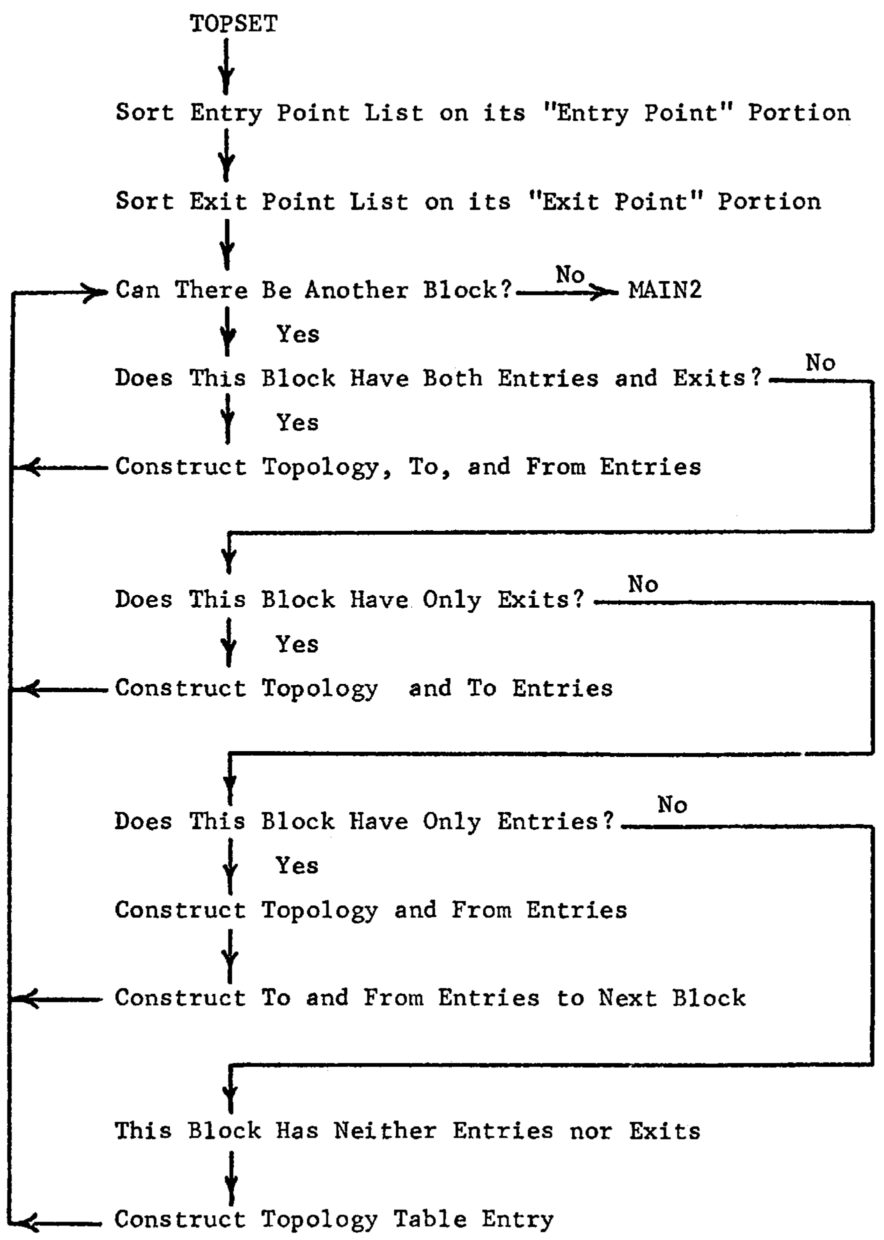

CHANGE
Are There More Result Locations?
No
MAIN2
Yes
Get Next Result Location
No
Is It a New Result Location?
Yes
Are There More Data or Storage List Entries?
No
Yes
Get Next Data or Storage List Entry
No
Is Result Location in Data or Storage Entry Locations?
Yes
Get Next Passive Reference Entry
No
Does "Instruction Cell" equal Result Location?
Yes
Set Changed Flag in "f" of Passive Reference

Figure A1.26 - TOPSET


TOPSET
Sort Entry Point List on its "Entry Point" Portion
Sort Exit Point List on its "Exit Point" Portion
Can There Be Another Block?
No
MAIN2
Yes
Does This Block Have Both Entries and Exits?
No
Yes
Construct Topology, To, and From Entries
Does This Block Have Only Exits?
No
Yes
Construct Topology and To Entries
Does This Block Have Only Entries?
No
Yes
Construct Topology and From Entries
Construct To and From Entries to Next Block
This Block Has Neither Entries nor Exits
Construct Topology Table Entry

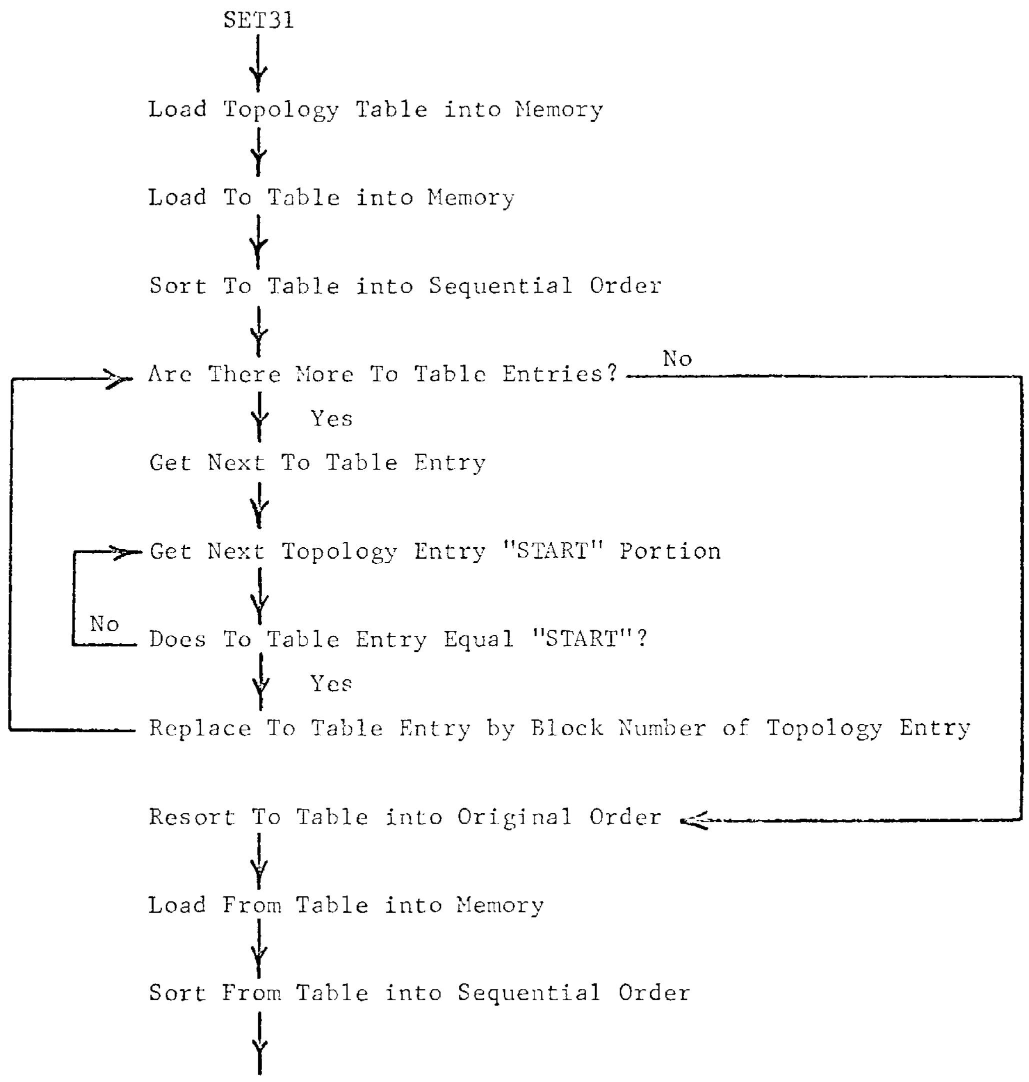
PHASE THREE

MAIN3 is the main program of Phase Three as shown in Figure 4.1. MAIN3 calls five subroutines which perform the required Data Reduction functions. SET31 reads the Control Tables into memory and converts the To and From Table contents from instruction locations to Block Numbers. CONECT checks the block interconnections and makes the required corrections. LOOP detects all program loops and flags both To and From Table loop closing branches. SET32 loads the Active and Passive Reference Lists into memory and constructs the active and passive entries in the Topology Table. LATEST determines the latest reference sets for each passive reference and stores the latest reference information in the Latest Reference and User Lists.

Figure A1.27 - MAIN3

$$\begin{tikzcd}
\text{MAIN3} \arrow[d] \\
\text{Read Control Tables into Memory (SET31)} \arrow[d] \\
\text{Check Block Interconnections (CONECT)} \arrow[d] \\
\text{Flag Program Loops (LOOP)} \arrow[d] \\
\text{Read Reference Lists into Memory (SET32)} \arrow[d] \\
\text{Find the Latest Reference (LATEST)} \arrow[d] \\
\text{MAIN4}
\end{tikzcd}$$

Figure A1.28 - SET31


SET31
Load Topology Table into Memory
Load To Table into Memory
Sort To Table into Sequential Order
Are There More To Table Entries?
No
Yes
Get Next To Table Entry
Get Next Topology Entry "START" Portion
No
Does To Table Entry Equal "START"?
Yes
Replace To Table Entry by Block Number of Topology Entry
Resort To Table into Original Order
Load From Table into Memory
Sort From Table into Sequential Order

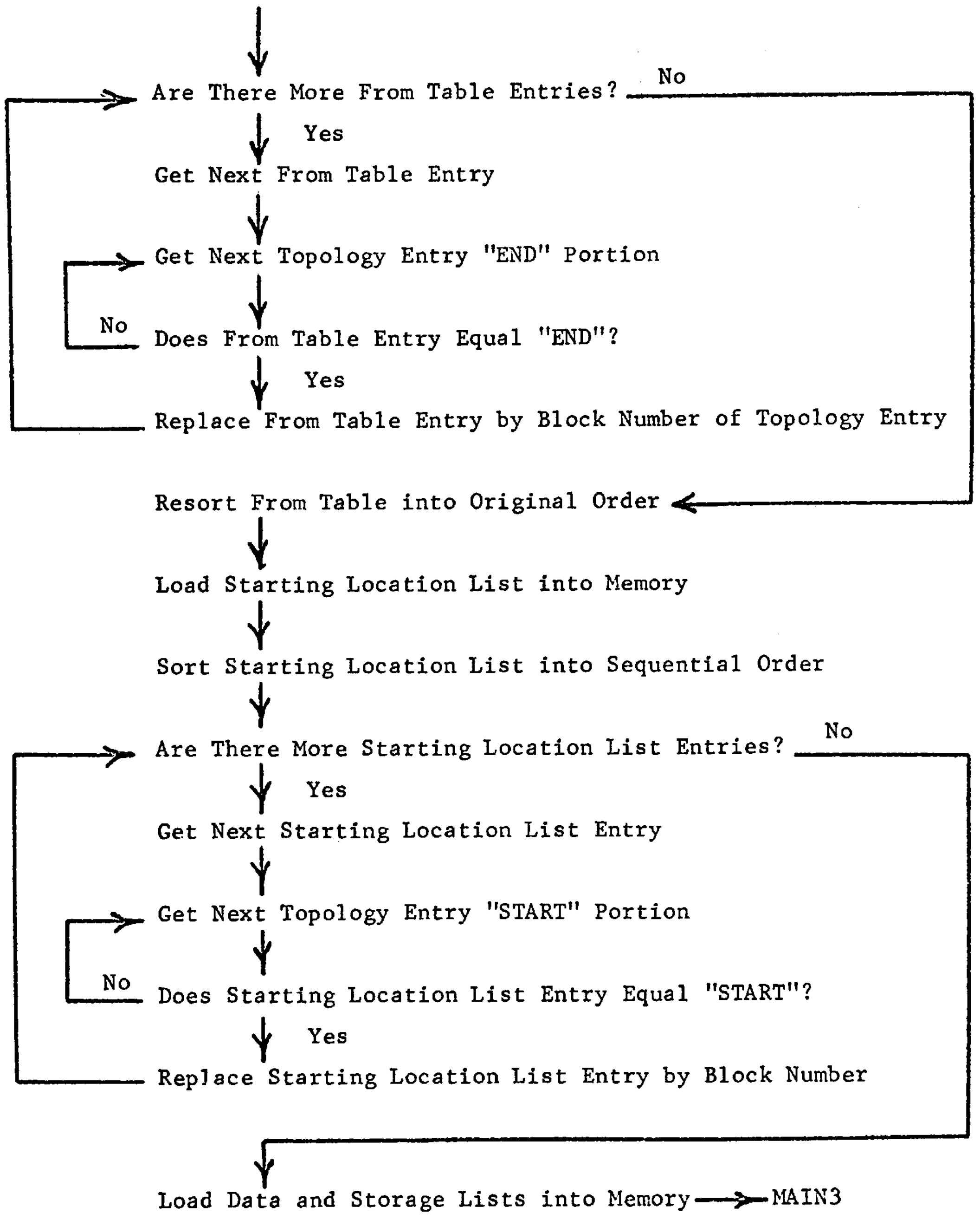
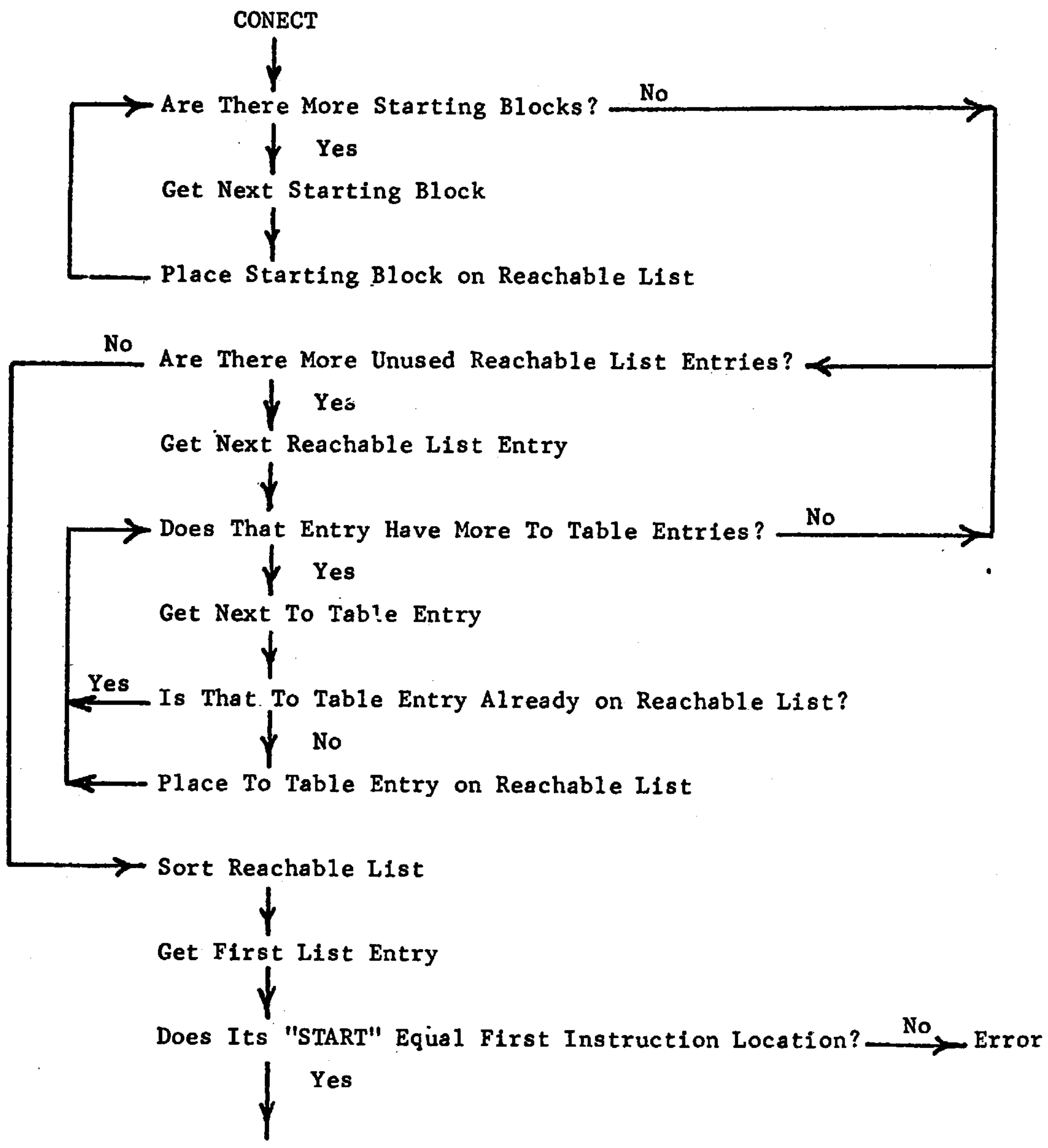

Are There More From Table Entries?
No
Yes
Get Next From Table Entry
Get Next Topology Entry "END" Portion
No
Does From Table Entry Equal "END"?
Yes
Replace From Table Entry by Block Number of Topology Entry
Resort From Table into Original Order
Load Starting Location List into Memory
Sort Starting Location List into Sequential Order
Are There More Starting Location List Entries?
No
Yes
Get Next Starting Location List Entry
Get Next Topology Entry "START" Portion
No
Does Starting Location List Entry Equal "START"?
Yes
Replace Starting Location List Entry by Block Number
Load Data and Storage Lists into Memory
MAIN3

Figure A1.29 - CONECT


CONECT
Are There More Starting Blocks?
No
Yes
Get Next Starting Block
Place Starting Block on Reachable List
No
Are There More Unused Reachable List Entries?
Yes
Get Next Reachable List Entry
Does That Entry Have More To Table Entries?
No
Yes
Get Next To Table Entry
Yes
Is That To Table Entry Already on Reachable List?
No
Place To Table Entry on Reachable List
Sort Reachable List
Get First List Entry
Does Its "START" Equal First Instruction Location?
No
Error
Yes

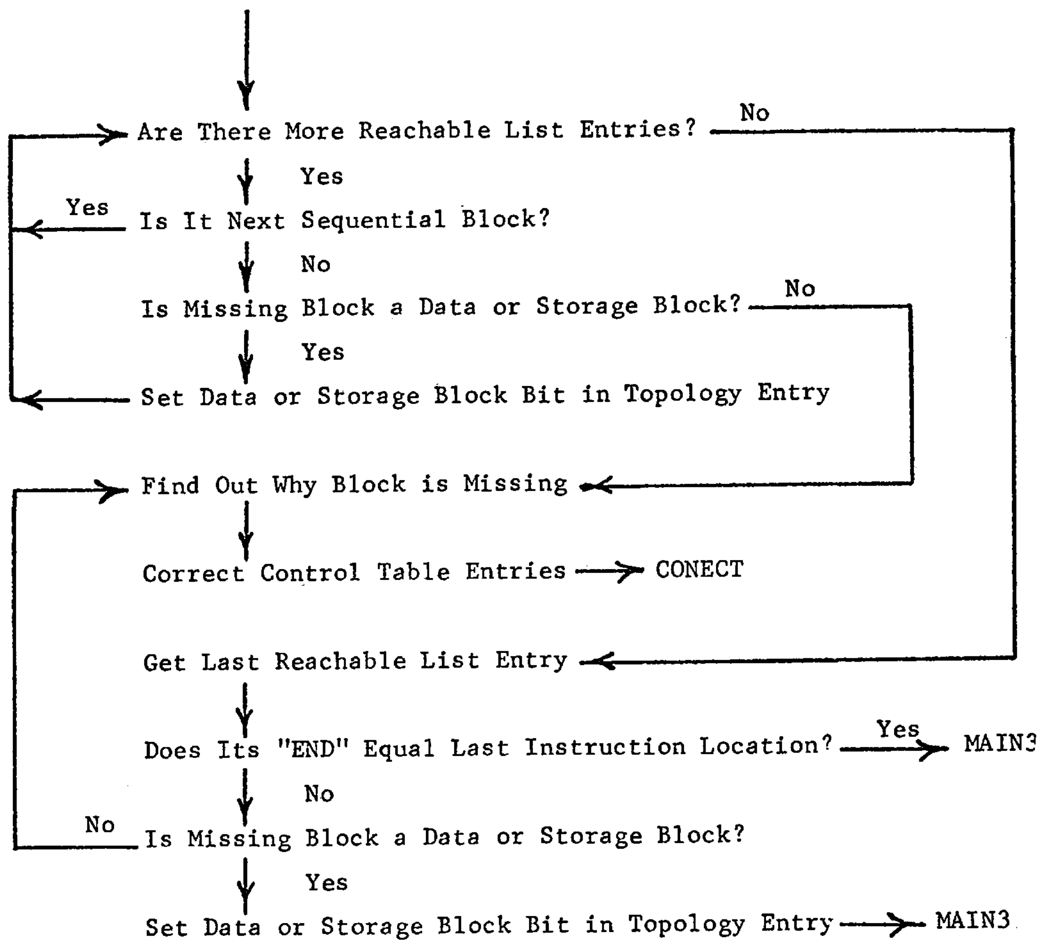

Are There More Reachable List Entries?
No
Yes
Yes
Is It Next Sequential Block?
No
Is Missing Block a Data or Storage Block?
No
Yes
Set Data or Storage Block Bit in Topology Entry
Find Out Why Block is Missing
Correct Control Table Entries
CONECT
Get Last Reachable List Entry
Does Its "END" Equal Last Instruction Location?
Yes
MAIN3
No
No
Is Missing Block a Data or Storage Block?
Yes
Set Data or Storage Block Bit in Topology Entry
MAIN3

Figure A1.30 - LOOP

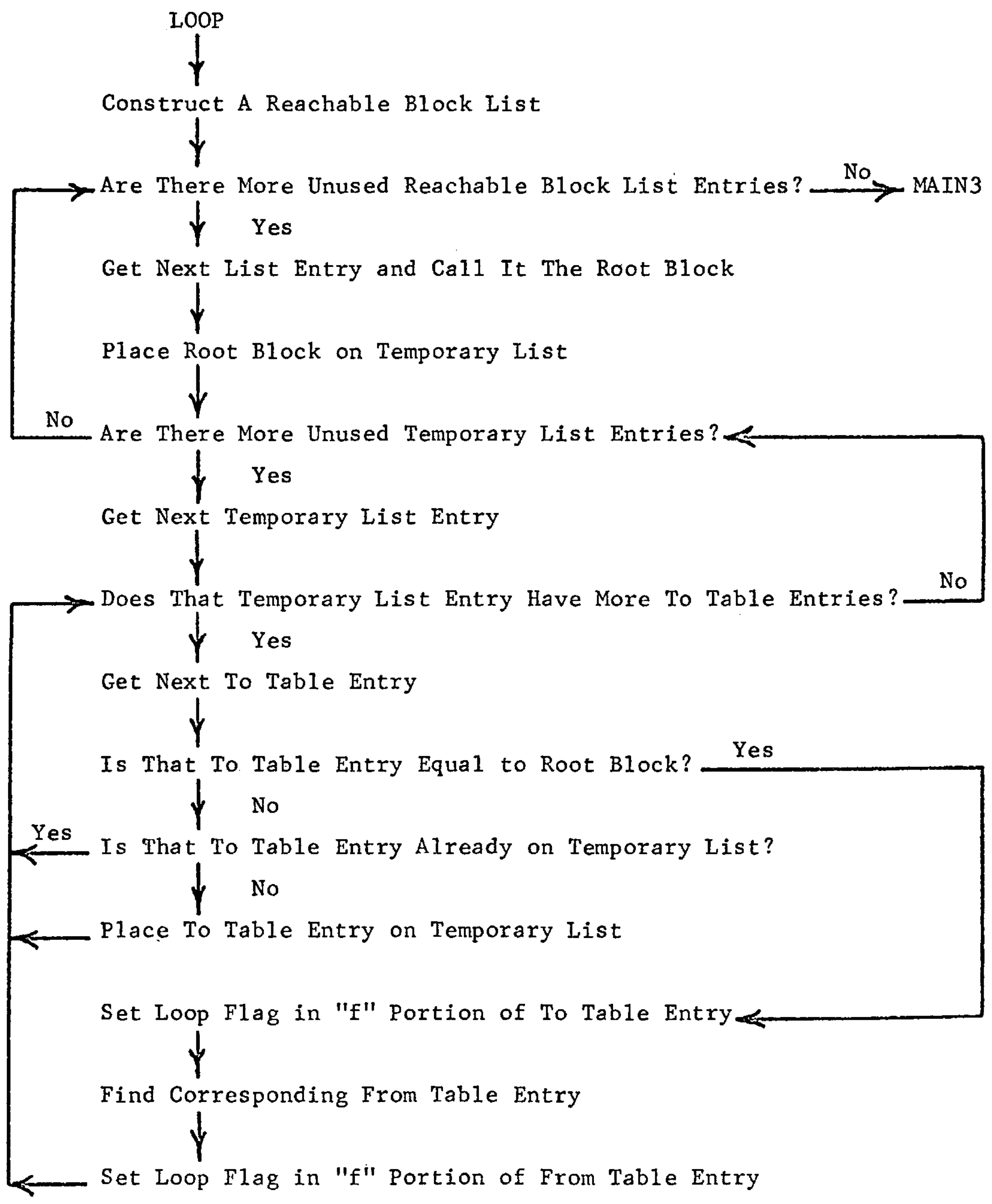

LOOP
Construct A Reachable Block List
Are There More Unused Reachable Block List Entries?
No
MAIN3
Yes
Get Next List Entry and Call It The Root Block
Place Root Block on Temporary List
No
Are There More Unused Temporary List Entries?
Yes
Get Next Temporary List Entry
Does That Temporary List Entry Have More To Table Entries?
No
Yes
Get Next To Table Entry
Is That To Table Entry Equal to Root Block?
Yes
No
Yes
Is That To Table Entry Already on Temporary List?
No
Place To Table Entry on Temporary List
Set Loop Flag in "f" Portion of To Table Entry
Find Corresponding From Table Entry
Set Loop Flag in "f" Portion of From Table Entry

Figure A1.31 - SET32

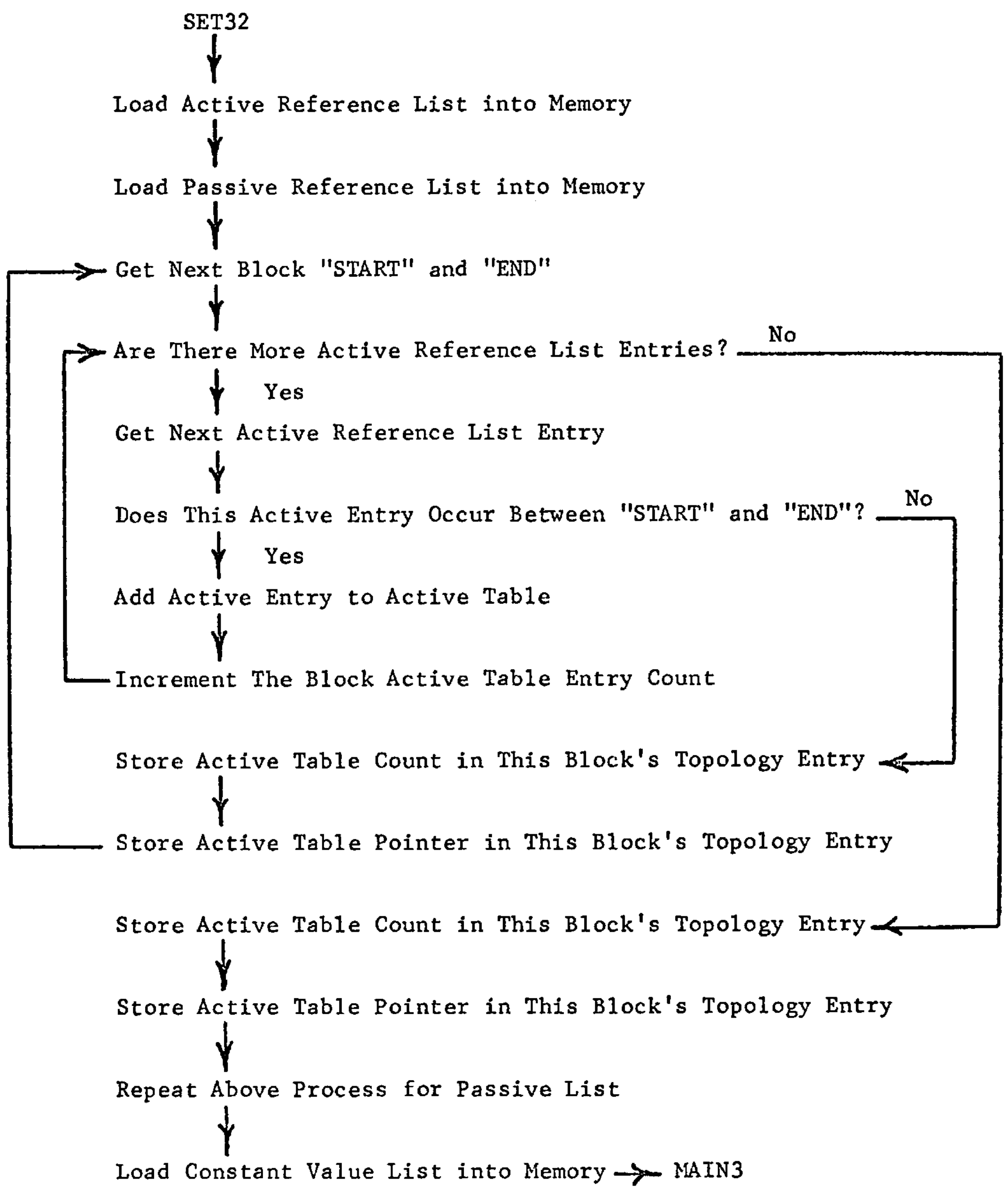
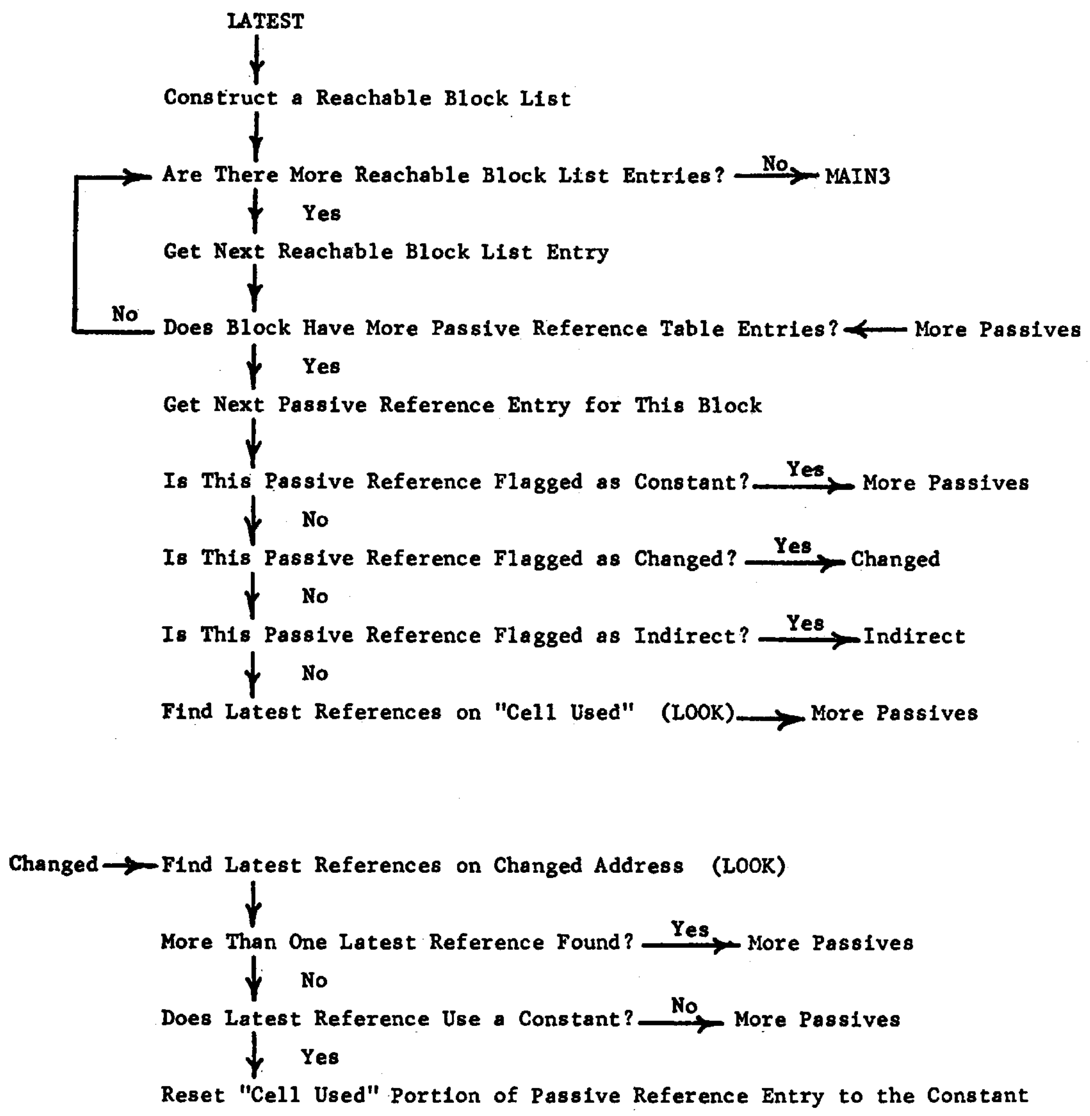
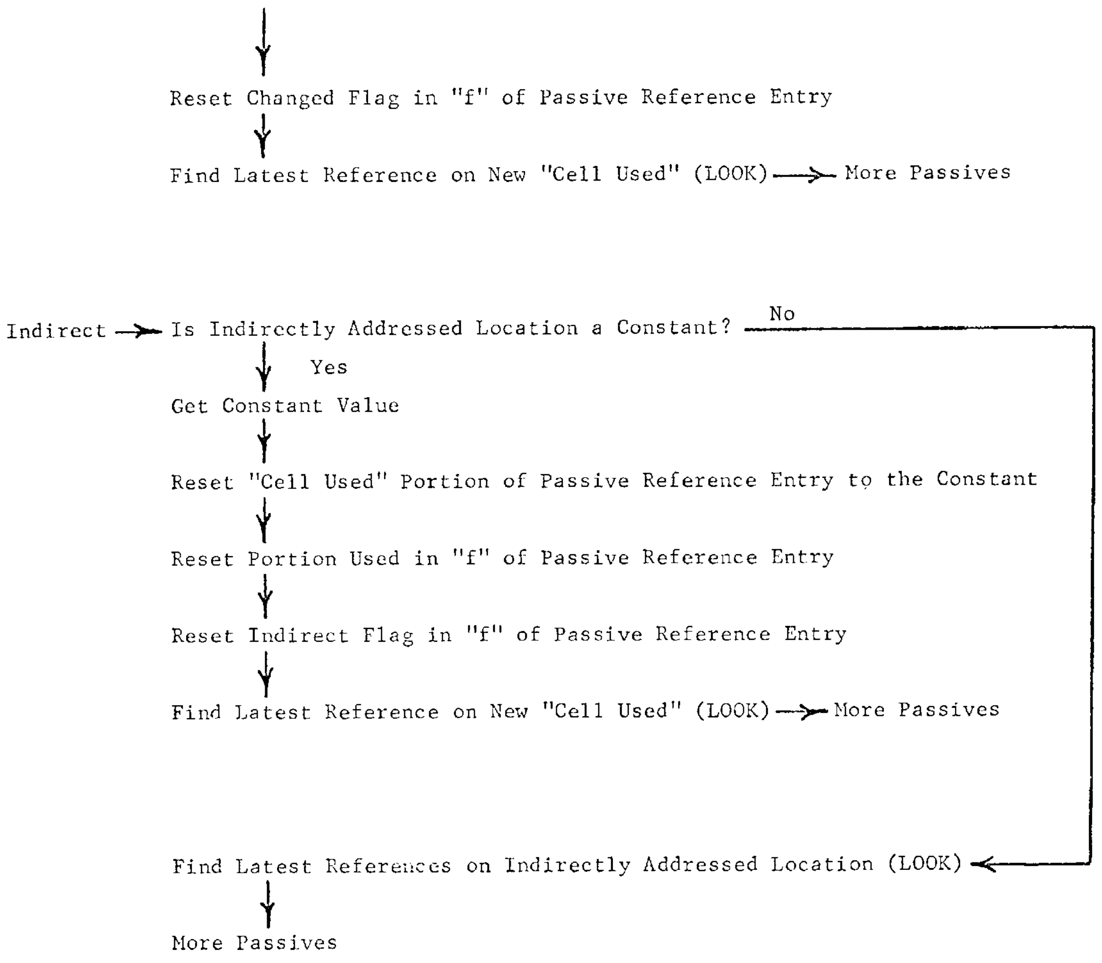
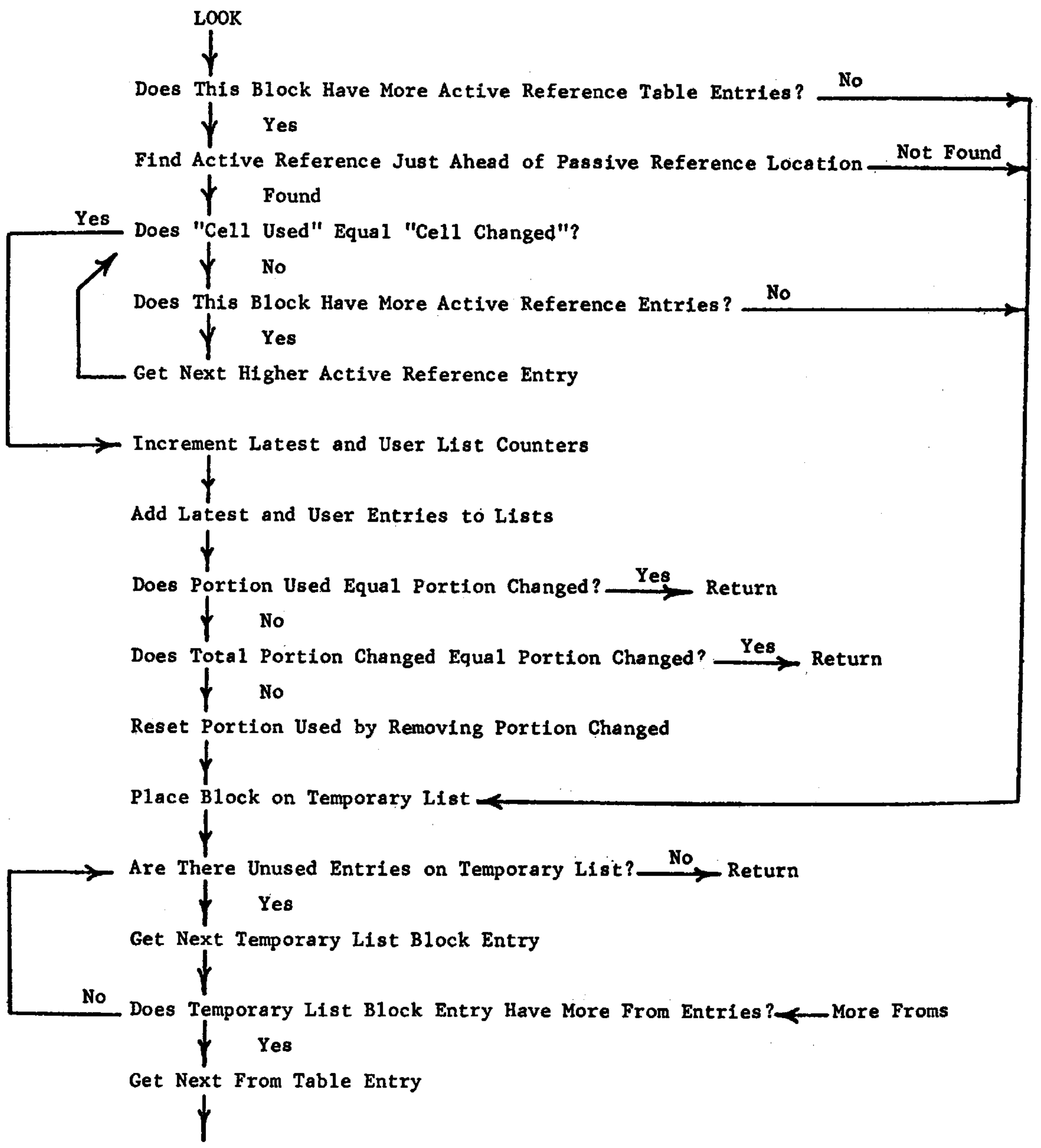
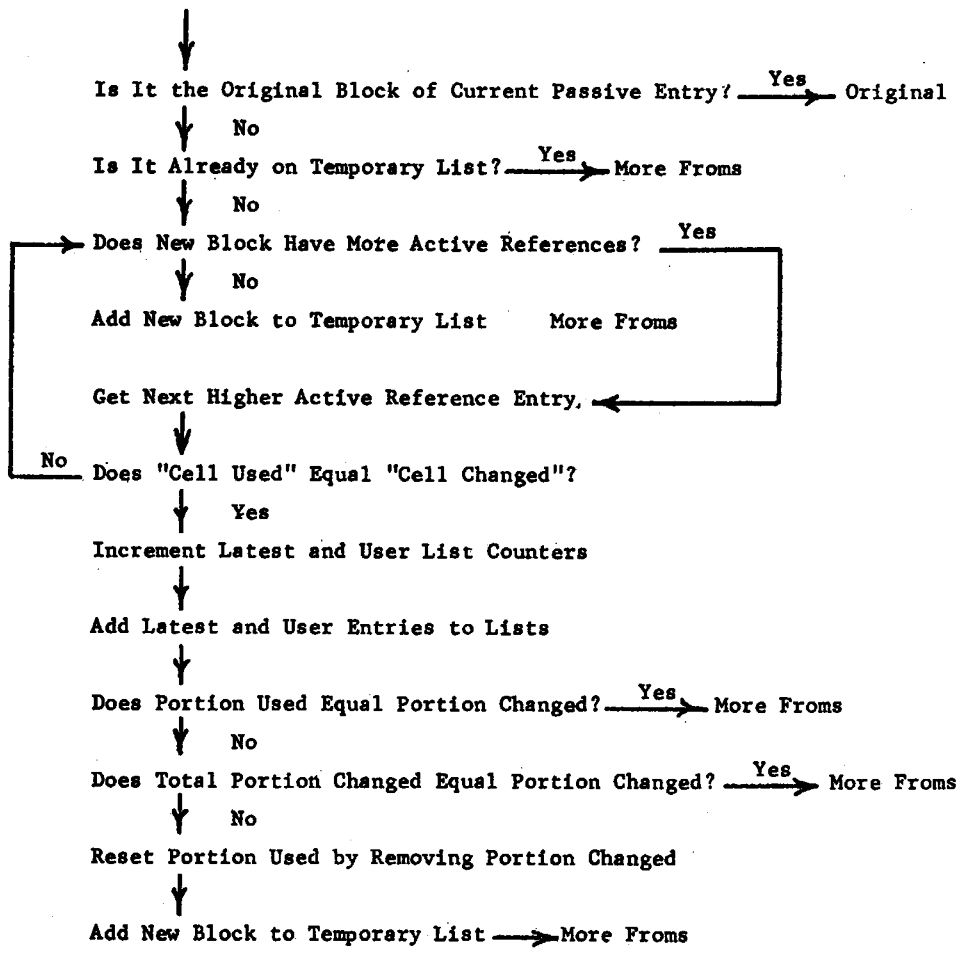

SET32
Load Active Reference List into Memory
Load Passive Reference List into Memory
Get Next Block "START" and "END"
Are There More Active Reference List Entries?
No
Yes
Get Next Active Reference List Entry
Does This Active Entry Occur Between "START" and "END"?
No
Yes
Add Active Entry to Active Table
Increment The Block Active Table Entry Count
Store Active Table Count in This Block's Topology Entry
Store Active Table Pointer in This Block's Topology Entry
Store Active Table Count in This Block's Topology Entry
Store Active Table Pointer in This Block's Topology Entry
Repeat Above Process for Passive List
Load Constant Value List into Memory
MAIN3

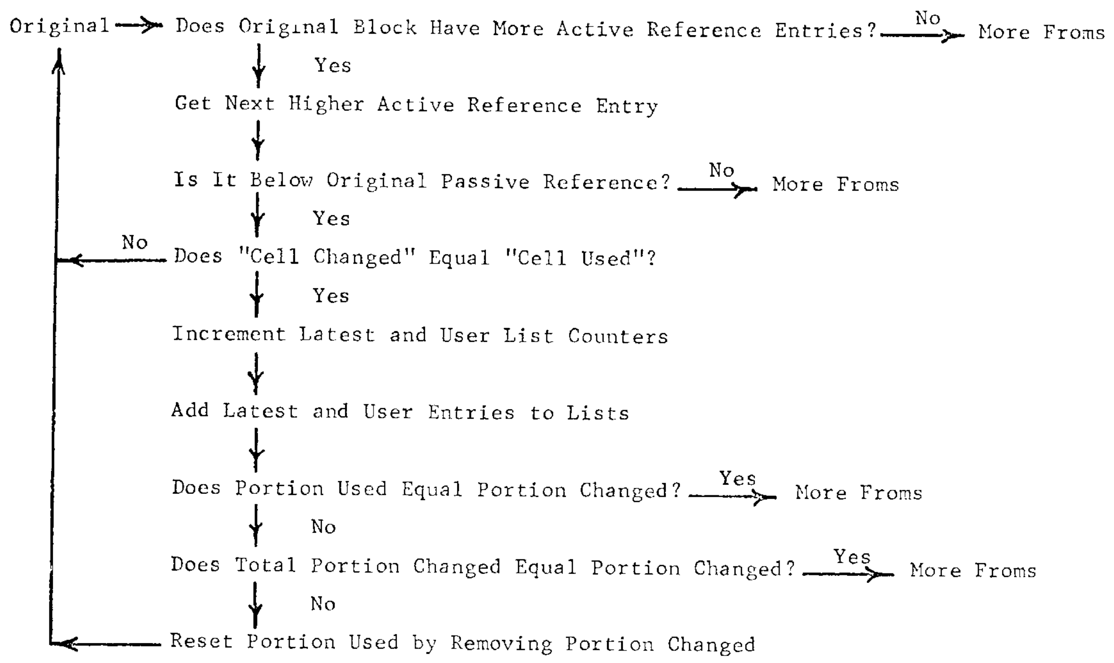
Figure A1.32 - LATEST


LATEST
Construct a Reachable Block List
Are There More Reachable Block List Entries?
No
MAIN3
Yes
Get Next Reachable Block List Entry
No
Does Block Have More Passive Reference Table Entries?
More Passives
Yes
Get Next Passive Reference Entry for This Block
Is This Passive Reference Flagged as Constant?
Yes
More Passives
No
Is This Passive Reference Flagged as Changed?
Yes
Changed
No
Is This Passive Reference Flagged as Indirect?
Yes
Indirect
No
Find Latest References on "Cell Used" (LOOK)
More Passives
Changed
Find Latest References on Changed Address (LOOK)
More Than One Latest Reference Found?
Yes
More Passives
No
Does Latest Reference Use a Constant?
No
More Passives
Yes
Reset "Cell Used" Portion of Passive Reference Entry to the Constant


Reset Changed Flag in "f" of Passive Reference Entry
Find Latest Reference on New "Cell Used" (LOOK)
More Passives
Indirect
Is Indirectly Addressed Location a Constant?
No
Yes
Get Constant Value
Reset "Cell Used" Portion of Passive Reference Entry to the Constant
Reset Portion Used in "f" of Passive Reference Entry
Reset Indirect Flag in "f" of Passive Reference Entry
Find Latest Reference on New "Cell Used" (LOOK)
More Passives
Find Latest References on Indirectly Addressed Location (LOOK)
More Passives


LOOK
Does This Block Have More Active Reference Table Entries?
No
Yes
Find Active Reference Just Ahead of Passive Reference Location
Not Found
Found
Yes
Does "Cell Used" Equal "Cell Changed"?
No
Does This Block Have More Active Reference Entries?
No
Yes
Get Next Higher Active Reference Entry
Increment Latest and User List Counters
Add Latest and User Entries to Lists
Does Portion Used Equal Portion Changed?
Yes
Return
No
Does Total Portion Changed Equal Portion Changed?
Yes
Return
No
Reset Portion Used by Removing Portion Changed
Place Block on Temporary List
Are There Unused Entries on Temporary List?
No
Return
Yes
Get Next Temporary List Block Entry
No
Does Temporary List Block Entry Have More From Entries?
More Froms
Yes
Get Next From Table Entry


Is It the Original Block of Current Passive Entry?
Yes
Original
No
Is It Already on Temporary List?
Yes
More Froms
No
Does New Block Have More Active References?
Yes
No
Add New Block to Temporary List
More Froms
Get Next Higher Active Reference Entry.
No
Does "Cell Used" Equal "Cell Changed"?
Yes
Increment Latest and User List Counters
Add Latest and User Entries to Lists
Does Portion Used Equal Portion Changed?
Yes
More Froms
No
Does Total Portion Changed Equal Portion Changed?
Yes
More Froms
No
Reset Portion Used by Removing Portion Changed
Add New Block to Temporary List
More Froms


Original
Does Original Block Have More Active Reference Entries?
No
More Froms
Yes
Get Next Higher Active Reference Entry
Is It Below Original Passive Reference?
No
More Froms
Yes
No
Does "Cell Changed" Equal "Cell Used"?
Yes
Increment Latest and User List Counters
Add Latest and User Entries to Lists
Does Portion Used Equal Portion Changed?
Yes
More Froms
No
Does Total Portion Changed Equal Portion Changed?
Yes
More Froms
No
Reset Portion Used by Removing Portion Changed

PHASE FOUR

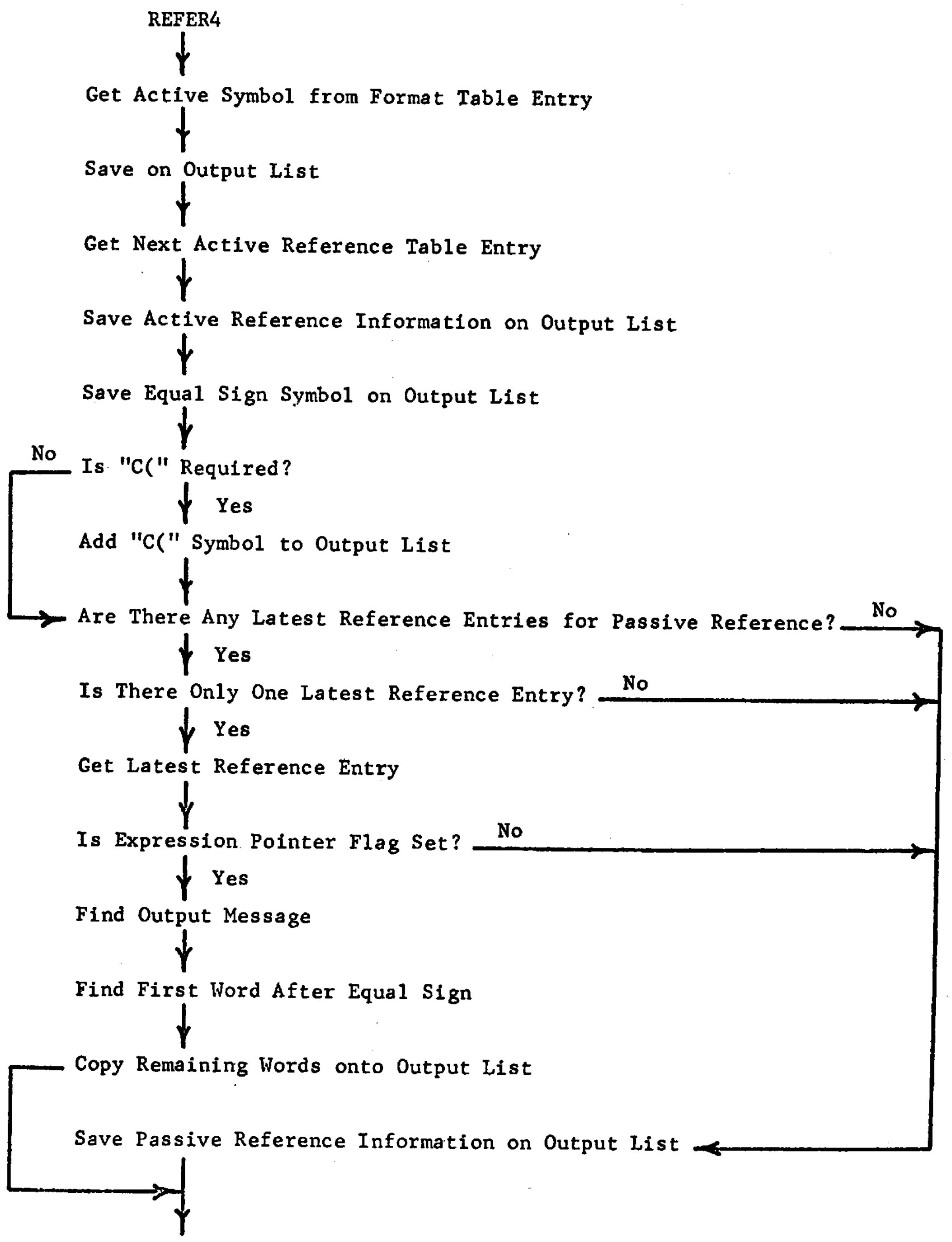
For programming purposes, Phase Four is divided into two parts. MAIN4 is the main program of the first part of Phase Four. MAIN4 calls two subroutines which generate the functional expressions. SET41 reads the various lists and tables into memory and constructs the Latest and User entries in the Topology Table. PERT first generates a reachable block list and then constructs a functional expression for each active reference. MAIN5 is the main program of the second part of Phase Four. MAIN5 calls two subroutines which produce the detailed output flowchart. SET51 reads the various lists into memory and sorts the Message Pointer List into sequential order. OUTPUT uses the Topology Table and the ordered Message Pointer List to produce the output flowchart.

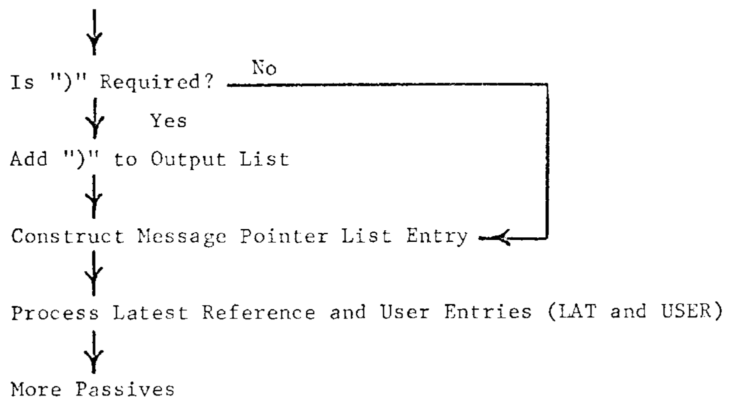
Figure A1.33 - MAIN4

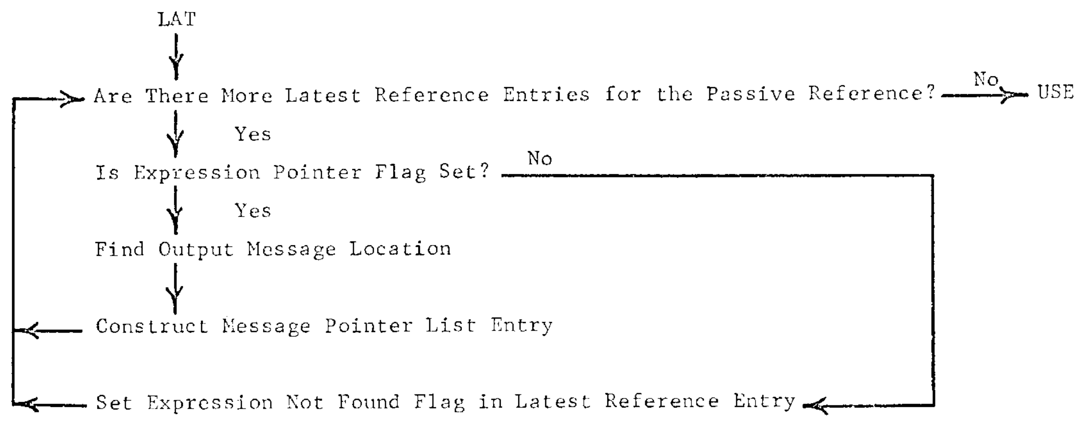
$$\begin{tikzcd}
\text{MAIN4} \arrow[d] \\
\text{Load The Required Tables (SET41)} \arrow[d] \\
\text{Construct Functional Expression (PERT)} \arrow[d] \\
\text{MAIN5}
\end{tikzcd}$$

Figure A1.34 - SET41

$$\begin{tikzcd}
\text{SET41} \arrow[d] & \\
\text{Load Topology Table into Memory} \arrow[d] & \\
\text{Load To Table into Memory} \arrow[d] & \\
\text{Load From Table into Memory} \arrow[d] & \\
\text{Load Starting Block List into Memory} \arrow[d] & \\
\text{Load Active Reference Table into Memory} \arrow[d] & \\
\text{Load Passive Reference Table into Memory} \arrow[d] & \\
\text{Load Latest Reference List into Memory} \arrow[d] & \\
\text{Construct Latest Reference Table} \arrow[d] & \\
\text{Load User List into Memory} \arrow[d] & \\
\text{Construct User Table} \arrow[r] & \text{MAIN5}
\end{tikzcd}$$


REFER4
Get Active Symbol from Format Table Entry
Save on Output List
Get Next Active Reference Table Entry
Save Active Reference Information on Output List
Save Equal Sign Symbol on Output List
Is "C(" Required?
No
Yes
Add "C(" Symbol to Output List
Are There Any Latest Reference Entries for Passive Reference?
No
Yes
Is There Only One Latest Reference Entry?
No
Yes
Get Latest Reference Entry
Is Expression Pointer Flag Set?
No
Yes
Find Output Message
Find First Word After Equal Sign
Copy Remaining Words onto Output List
Save Passive Reference Information on Output List


Is ")" Required?
No
Yes
Add ")" to Output List
Construct Message Pointer List Entry
Process Latest Reference and User Entries (LAT and USER)
More Passives


LAT
Are There More Latest Reference Entries for the Passive Reference?
No
USE
Yes
Is Expression Pointer Flag Set?
No
Yes
Find Output Message Location
Construct Message Pointer List Entry
Set Expression Not Found Flag in Latest Reference Entry

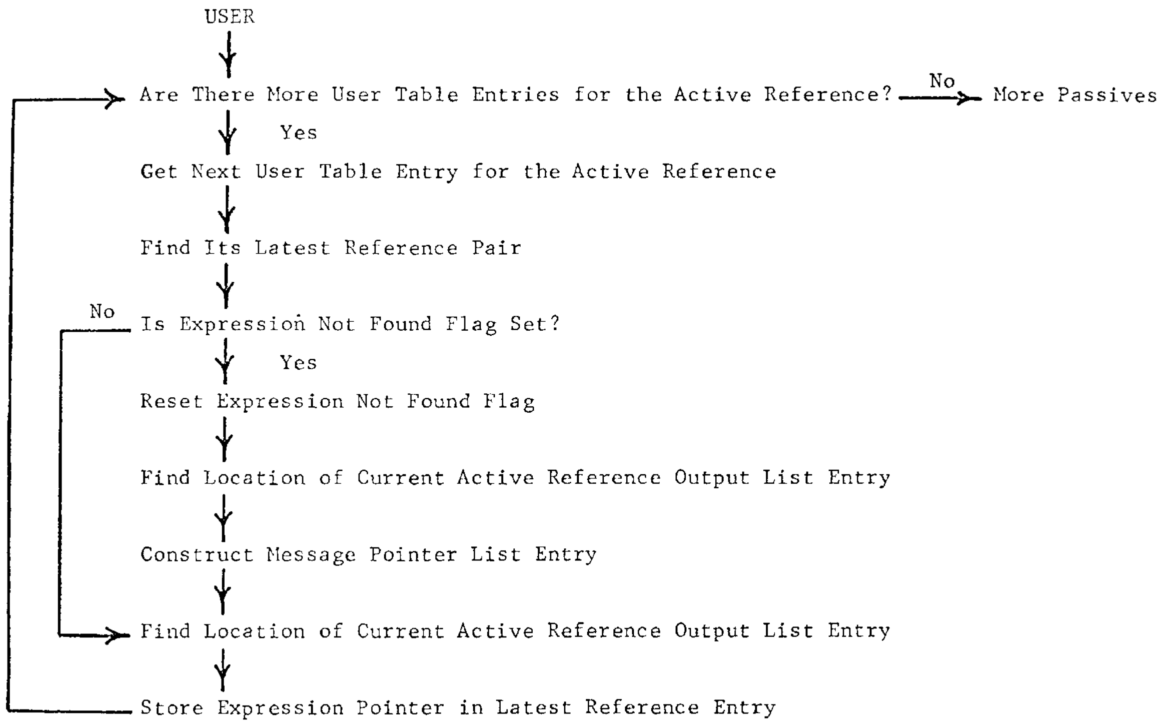

USER
Are There More User Table Entries for the Active Reference?
No
More Passives
Yes
Get Next User Table Entry for the Active Reference
Find Its Latest Reference Pair
No
Is Expression Not Found Flag Set?
Yes
Reset Expression Not Found Flag
Find Location of Current Active Reference Output List Entry
Construct Message Pointer List Entry
Find Location of Current Active Reference Output List Entry
Store Expression Pointer in Latest Reference Entry

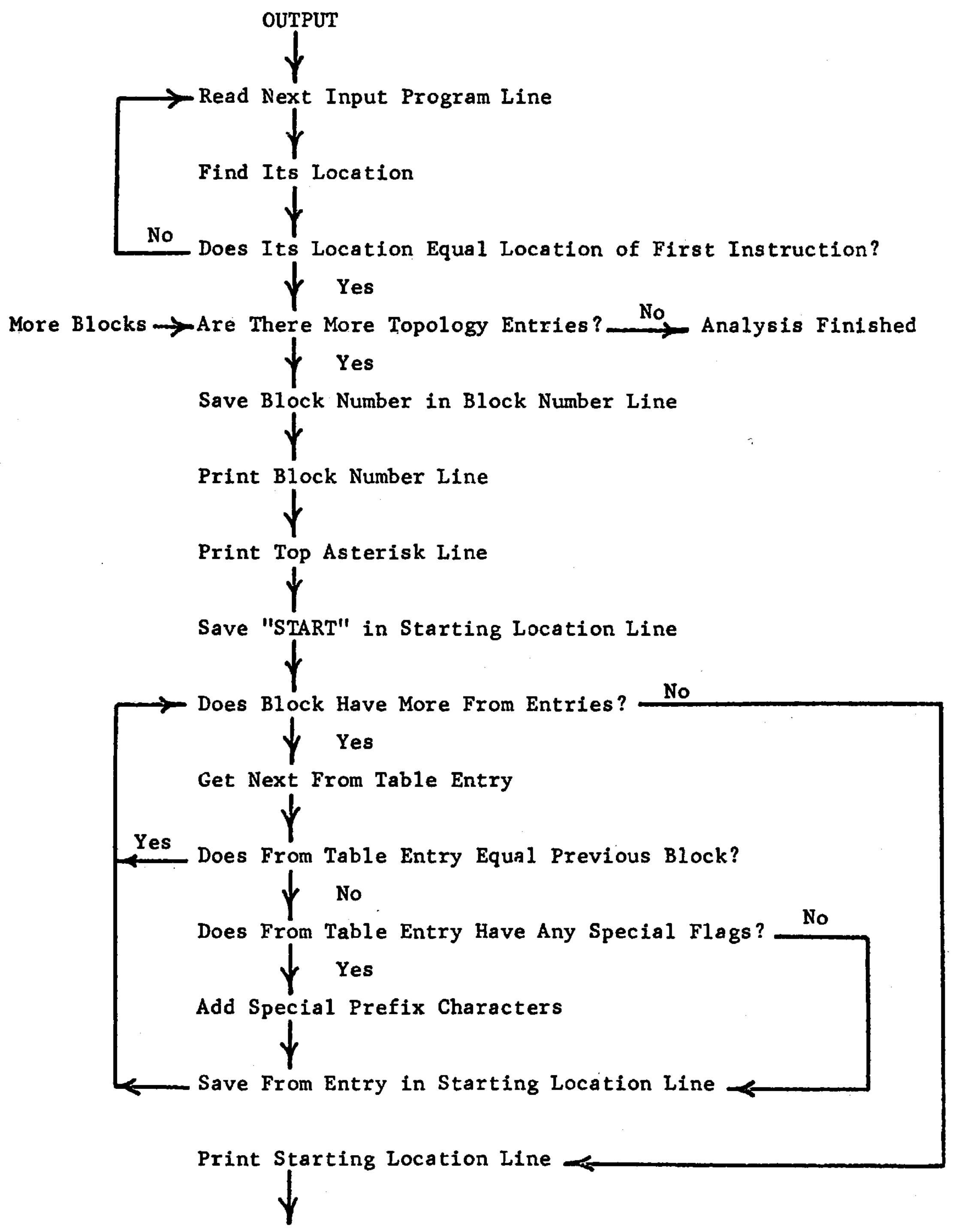
Figure A1.36 - MAIN5

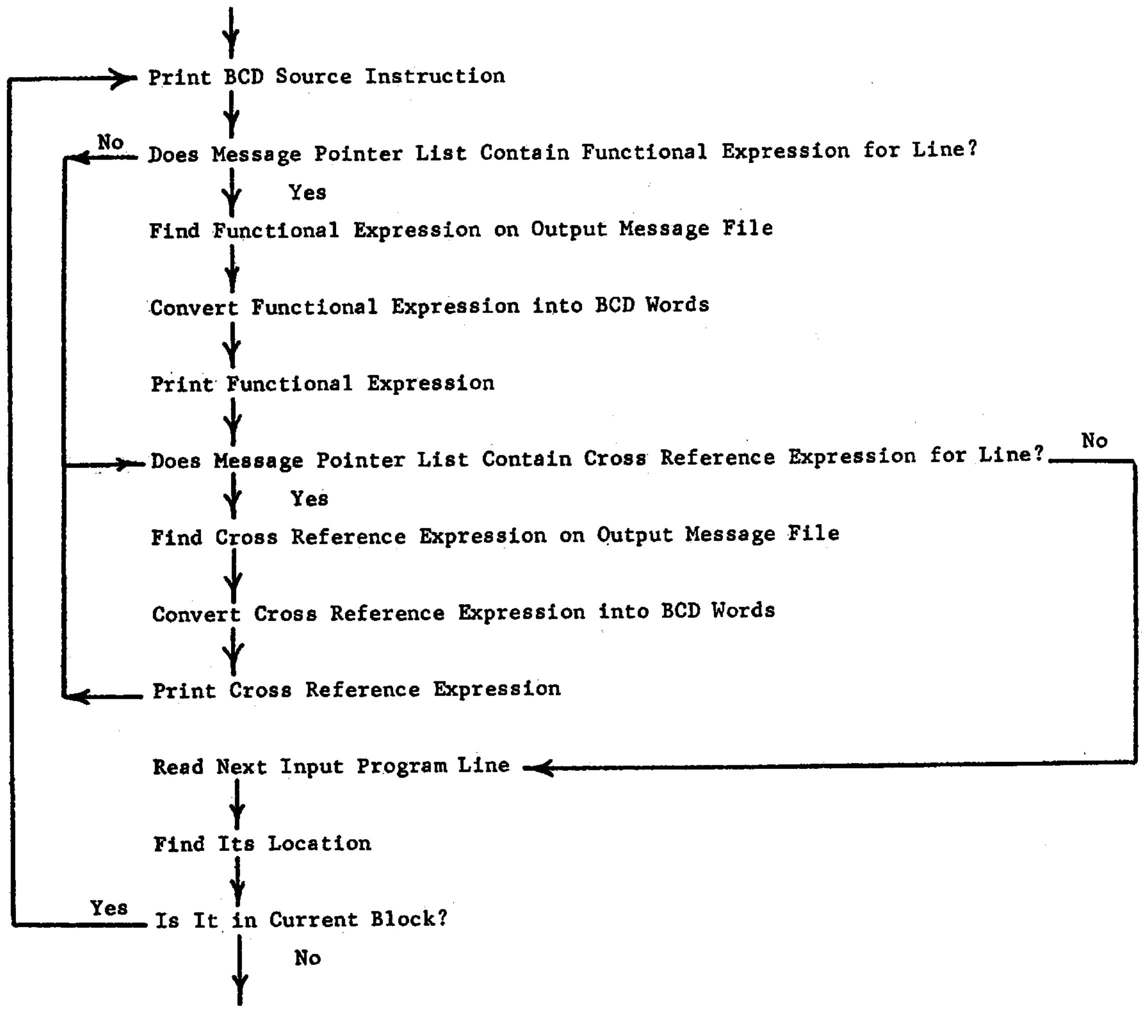
$$\begin{tikzcd}
\text{MAIN5} \arrow[d] \\
\text{Load The Required Tables (SET51)} \arrow[d] \\
\text{Print Output Flowchart (OUTPUT)}
\end{tikzcd}$$

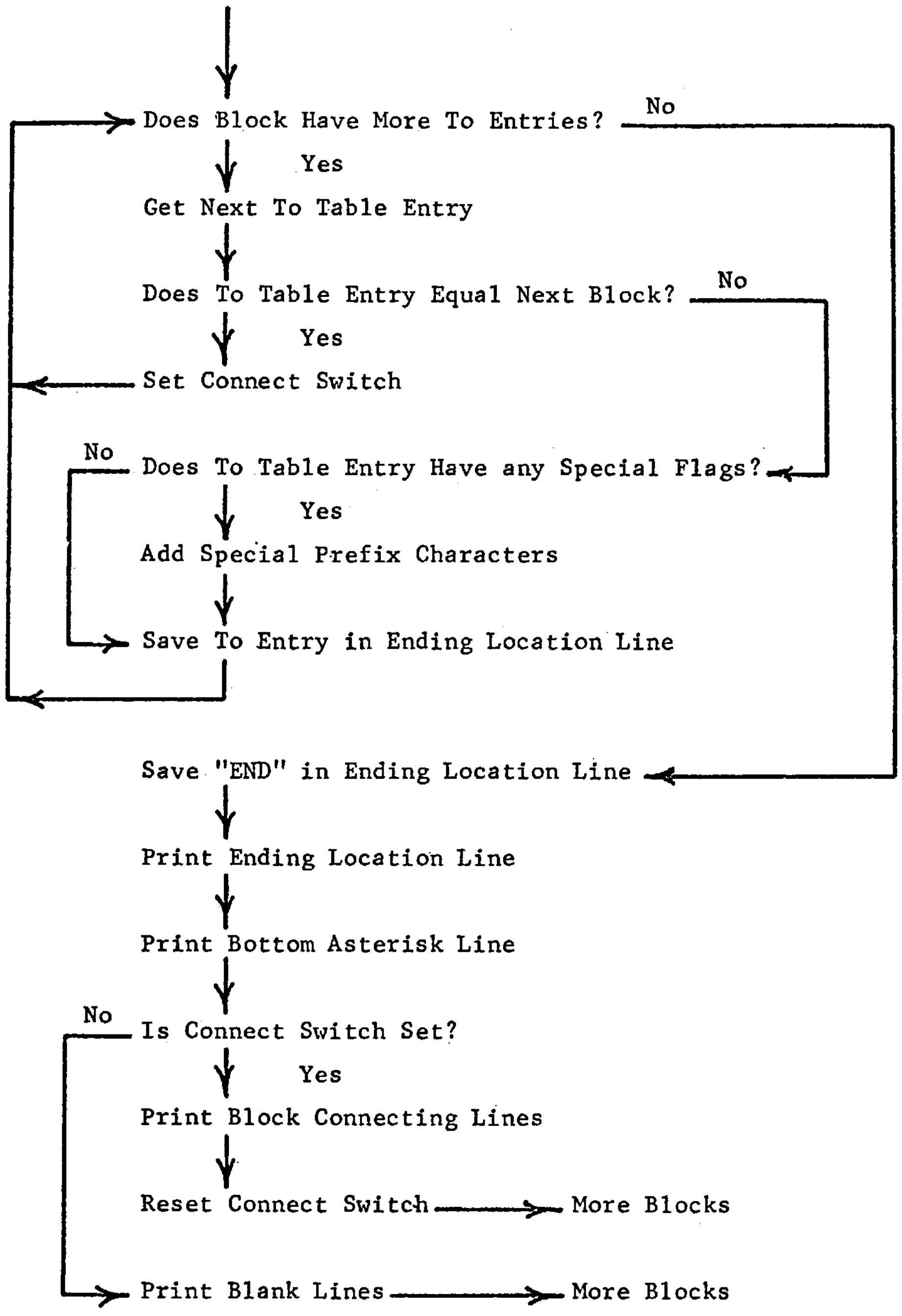
Figure A1.37 - SET51

$$\begin{tikzcd}
\text{SET51} \arrow[d] \\
\text{Load Topology Table into Memory} \arrow[d] \\
\text{Load To Table into Memory} \arrow[d] \\
\text{Load From Table into Memory} \arrow[d] \\
\text{Load Starting Block List into Memory} \arrow[d] \\
\text{Load Message Pointer List into Memory} \arrow[d] \\
\text{Sort Message Pointer List into Sequential Order} \arrow[d] \\
\text{Load Output Message File into Memory} \arrow[d] \\
\text{Load Symbol Table into Memory} \arrow[d] \\
\text{Sort Symbol Table into Sequential Order} \arrow[d] \\
\text{Load Constant Value List into Memory} \arrow[d] \\
\text{Load Data List into Memory} \arrow[d] \\
\text{Load Storage List into Memory} \arrow[r] & \text{MAIN5}
\end{tikzcd}$$

Figure A1.38 - OUTPUT


OUTPUT
Read Next Input Program Line
Find Its Location
No
Does Its Location Equal Location of First Instruction?
Yes
More Blocks
Are There More Topology Entries?
No
Analysis Finished
Yes
Save Block Number in Block Number Line
Print Block Number Line
Print Top Asterisk Line
Save "START" in Starting Location Line
Does Block Have More From Entries?
No
Yes
Get Next From Table Entry
Yes
Does From Table Entry Equal Previous Block?
No
Does From Table Entry Have Any Special Flags?
No
Yes
Add Special Prefix Characters
Save From Entry in Starting Location Line
Print Starting Location Line


Print BCD Source Instruction
Does Message Pointer List Contain Functional Expression for Line?
No
Yes
Find Functional Expression on Output Message File
Convert Functional Expression into BCD Words
Print Functional Expression
Does Message Pointer List Contain Cross Reference Expression for Line?
No
Yes
Find Cross Reference Expression on Output Message File
Convert Cross Reference Expression into BCD Words
Print Cross Reference Expression
Read Next Input Program Line
Find Its Location
Is It in Current Block?
Yes
No



Does Block Have More To Entries?
No
Yes
Get Next To Table Entry
Does To Table Entry Equal Next Block?
No
Yes
Set Connect Switch
No
Does To Table Entry Have any Special Flags?
Yes
Add Special Prefix Characters
Save To Entry in Ending Location Line
Save "END" in Ending Location Line
Print Ending Location Line
Print Bottom Asterisk Line
No
Is Connect Switch Set?
Yes
Print Block Connecting Lines
Reset Connect Switch
More Blocks
Print Blank Lines
More Blocks

APPENDIX TWO

FLOWCHARTS OF ACTUAL PROGRAMS

One standard question has been, "Can the Analysis Program analyze itself?" The purpose of this appendix is to display flowcharts of analysis subroutines produced automatically by the analysis program. In general, the Topological Flowcharts are accurate, while the Detailed Flowcharts are incomplete due to unprogrammed functional generation subroutines.

Figure A2.1a shows the listing of a subroutine which converts the binary number contained in the logical AC into a BCD number with leading blanks. Figure A2.1b displays the Topological Flowchart of the conversion program while Figure A2.1c displays the Detailed Flowchart.

Figure A2.1a - A Binary to BCD Conversion Subroutine

```
            ENTRY     BCD          SUBROUTINE TO CONVERT THE LOGICAL AC
*                                  TO A BCD NUMBER FROM AN OCTAL NUMBER
*                                  WITH LEADING B.
*
BCD         SXA       X4,4         SAVE THE RETURN ADDRESS
            XCL
            ROL       21           LEFT JUSTIFY THE FIVE DIGIT NUMBER
*
            AXT       5,4          THERE CAN BE FIVE DIGITS
S1          STQ       SMQ          SAVE THE REMAINDER OF THE NUMBER
            ZAC                    Z THE AC
            LGL       3            GET THE NEXT OCTAL DIGIT
            TNZ       S2           IF NON-Z, ADD DIGIT TO END OF AC
            TIX       S1,4,1       BYPASS LEADING ZS
            CAL       Z            GET A BCD ZERO
            TRA       X4
S2          CAL       B            INITIALIZE AC AS BLANKS
S3          LDQ       SMQ          RESTORE MQ FOR FIRST NON-Z DIGIT
S4          ALS       3            PICK UP THREE Z BITS
            LGL       3            PICK UP NEXT DIGIT
            TIX       S4,4,1       DECREASE DIGIT COUNT
*
X4          AXT       **,4         RESTORE RETURN ADDRESS
            TRA       1,4          RETURN
*
SMQ         PZE       0
Z           BCI       1,     0
B           BCI       1,
            END
```

Figure A2.1b - The Topological Flowchart for Figure A2.1a

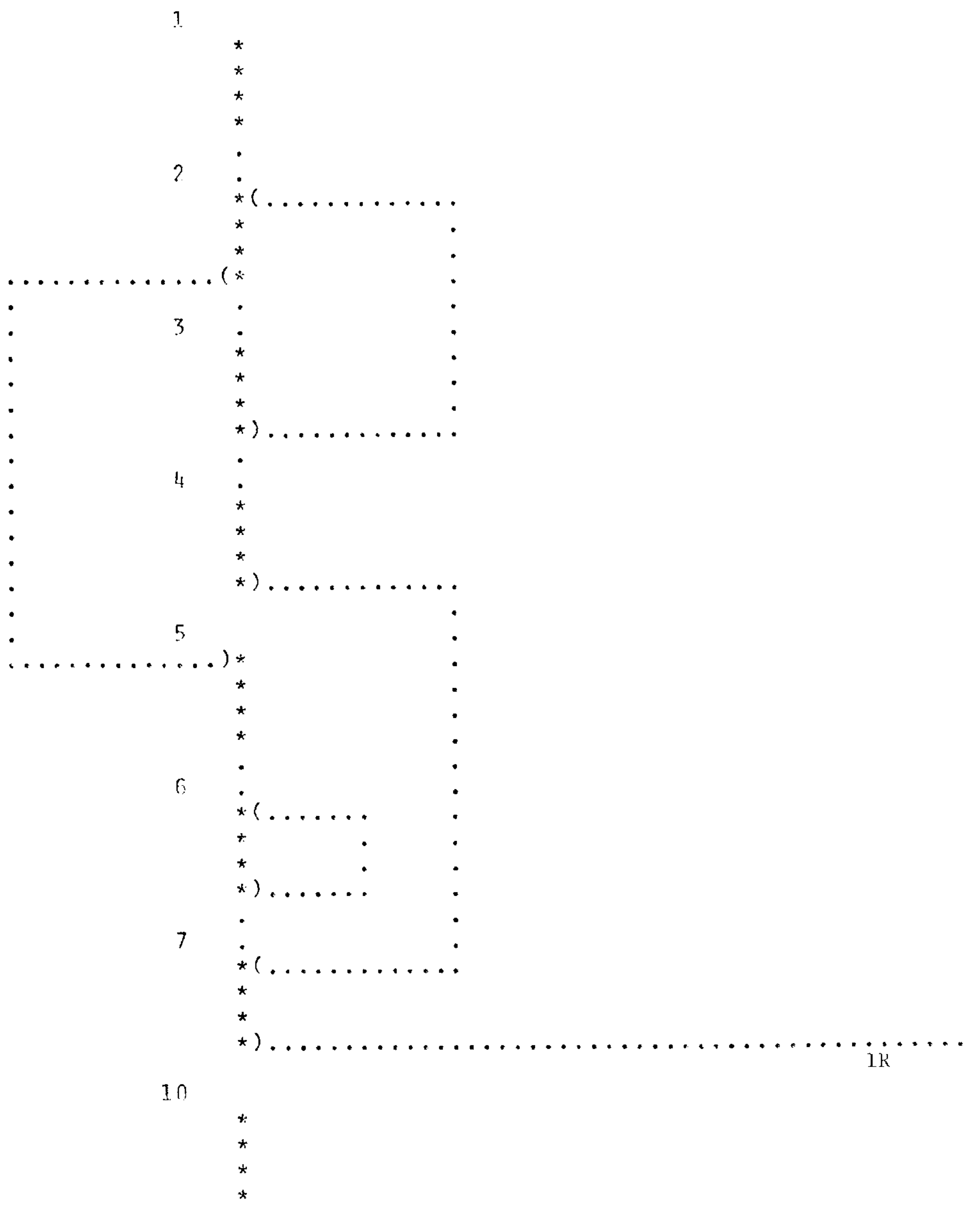


Figure A2.1c - The Detailed Flowchart for Figure A2.1a

```
                                                           1
                                             ***********************
                                         2   *                     *
02 BCD SYA      X4,4                         *     A/Y4=IY4        *
03      XCL                                  *                     *
04      RQL      21                          *                     *
05      AXT      5,4                         *     IY4=A/77400400005*
                                         5   *                     *
                                             ***********************
                                                         .
                                                         .
                                                         .
                                                   2     .
                                             ***********************
                                         6   *                     *(...   13
06 S1   STQ      SMQ                         *     SMQ=MQ          *
07      ZAC                                  *                     *
10      LGL      3                           *                     *
11      TNZ      S2                          *                     *
                                        11   *                     *)...   5
                                             ***********************
                                                         .
                                                         .
                                                         .
                                                   3     .
                                             ***********************
                                        12   *                     *
12      TIX      S1,4,1                      *                     *
                                        12   *                     *)...   12
                                             ***********************
                                                         .
                                                         .
                                                         .
                                                   4     .
                                             ***********************
                                        13   *                     *
13      CAL      Z                           *     AC=606060606000 *
14      TRA      Y4                          *                     *
                                        14   *                     *)...   7
                                             ***********************


                                                   5
                                             ***********************
                                        15   *                     *(...   2
15 S2   CAL      B                           *     AC=606060606060 *
16 S3   LDQ      SMQ                         *     MQ=MQ           *       6  SMQ=MQ
                                        16   *                     *
                                             ***********************
                                                         .
                                                         .
```

```
                                    6      .
                               ************************
                          17   *                      *(...  L6
17 S4   ALS      3             *                      *
20      LGL      3             *                      *
21      TIX      S4,4,1        *                      *
                          21   *                      *)...  L6
                               ************************
                                           .
                                           .
                                           .
                                    7      .
                               ************************
                          22   *                      *(...  4
22 X4   AXT      **,4          *     IX4=A/IX4        *      2 A/X4=IX4
23      TRA      1,4           *                      *
                          23   *                      *)...  IR
                               ************************


                                   10
                               ************************
                          24   *                      *
24 SMQ  PZE      0             *                      *
25 Z    BCI      1,     0      *                      *
26 B    BCI      1,            *                      *
                          26   *                      *
                               ************************
```

Figure A2.2a shows the listing of a program which sorts a table on its address portion and then within the same address by tag. First, the program interchanges the address and tag portions of each entry. Second, the program calls a binary sort routine, DSRT18, to perform the sort. Third, the program returns the addresses and tags to their original positions. Figure A2.2b displays the Topological Flowchart of the program.

Figure A2.2a - An Address and Tag Sort Subroutine

```
           ENTRY      SRTTAG       SUBROUTINE TO SORT ON ADDRESS,
*                                  THEN INTERNALLY ON TAG
*
SRTTAG     SXA        .IX1,1
           SXA        .IX2,2
           SXA        .IX4,4
*
           CAL        1,4          GET ADDRESS OF TOP ADDRESS
           STA        .A4
           CAL        2,4          GET ADDRESS OF COUNT ADDRESS
           STA        .A5
*
           CAL*       1,4          GET TOP ADDRESS
           STA        .A1
           STA        .A2
           STA        .A3
           STA        .A6
           STA        .A7
           STA        .A8
           CAL*       2,4          GET THE COUNT
           STA        COUNT
           PAX        ,1
*
.A1        CAL        **,1         GET NEXT WORD
           PAX        ,2           SAVE ADDRESS
           LGR        18           SAVE TAG
           PXA        ,2           GET ADDRESS
           LGL        3            PICKUP TAG
.A2        STT        **,1         SAVE LAST 18 BITS
.A3        STA        **,1
           TIX        .A1,1,2      ARE THERE MORE WORDS
*
           TSX        $DSRT18,4    SORT ON LAST 18 BITS
.A4        PZE        **
.A5        PZE        **
*
           ZAC                     ZERO THE MQ
           XCL
           LXA        COUNT,1      GET THE COUNT
.A6        CAL        **,1         GET NEXT WORD
           ALS        15
.A7        STT        **,1         SAVE OLD TAG
           PDX        ,2
           PXA        ,2
.A8        STA        **,1         SAVE OLD ADDRESS
           TIX        .A6,1,2      ARE THERE MORE WORDS
*
.IX1       AXT        **,1
.IX2       AXT        **,2
.IX4       AXT        **,4
           TRA        3,4
*
COUNT      PZE        0
           END
```

Figure A2.2b - The Topological Flowchart for Figure A2.2a

```
1
      *
      *
      *
      *
      .
2     .
      *(.......
      *       .
      *       .
      *).......
      .
3     .
      *
      *
      *
      *)..................................................E5

4
      *
      *
      *
      *

5
      *(..................................................E3
      *
      *
      *
      .
6     .
      *(.......
      *       .
      *       .
      *).......
      .
7     .
      *
      *
      *
      *)..................................................IR

10
      *
      *
      *
      *
```

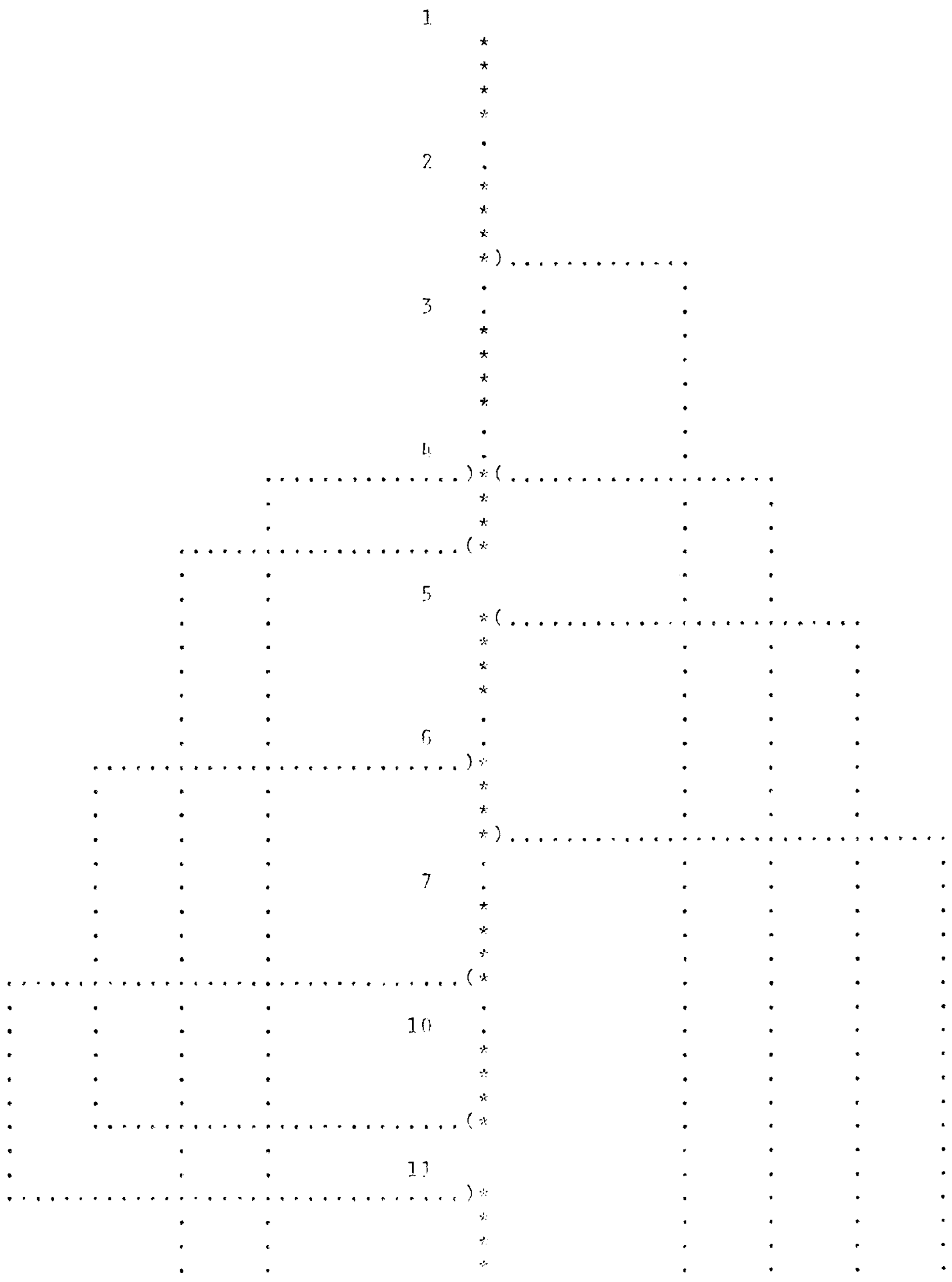
Figure A2.3a shows the listing of the DSRT18 program which performs a binary sort using only the right-hand eighteen bits. Figure A2.3b displays the Topological Flowchart of the sort program. While the DSRT18 subroutine was being analyzed, the CONECT subroutine found that there was an instruction just above location SORT31 in DSRT18 which could not be reached. This unreachable instruction turned out to be extraneous and must have been inserted while the program was being prepared for input to the computer. After the extra instruction was removed, the analysis program ran to completion.

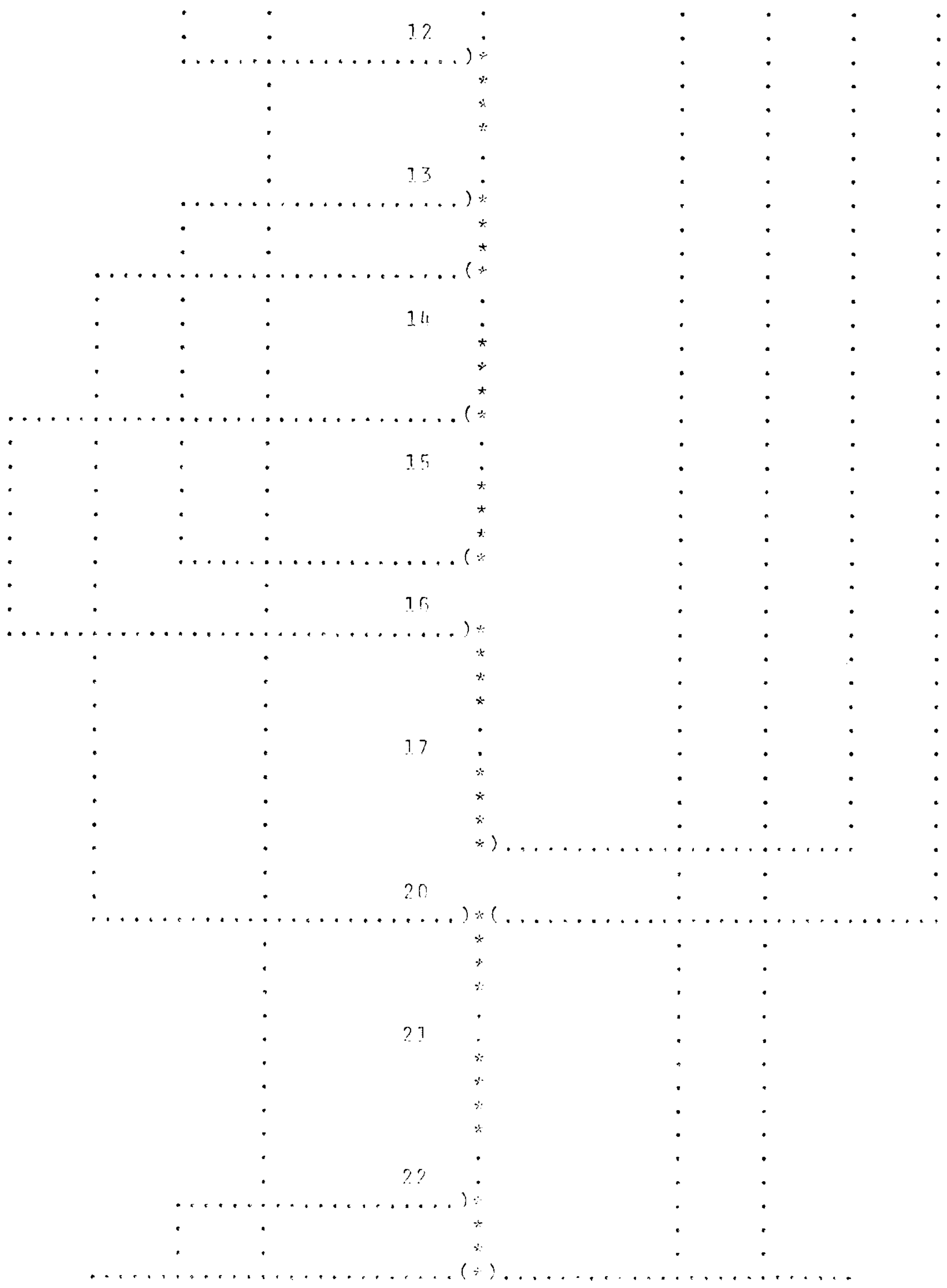
Figure A2.3a - A Binary Sort Subroutine

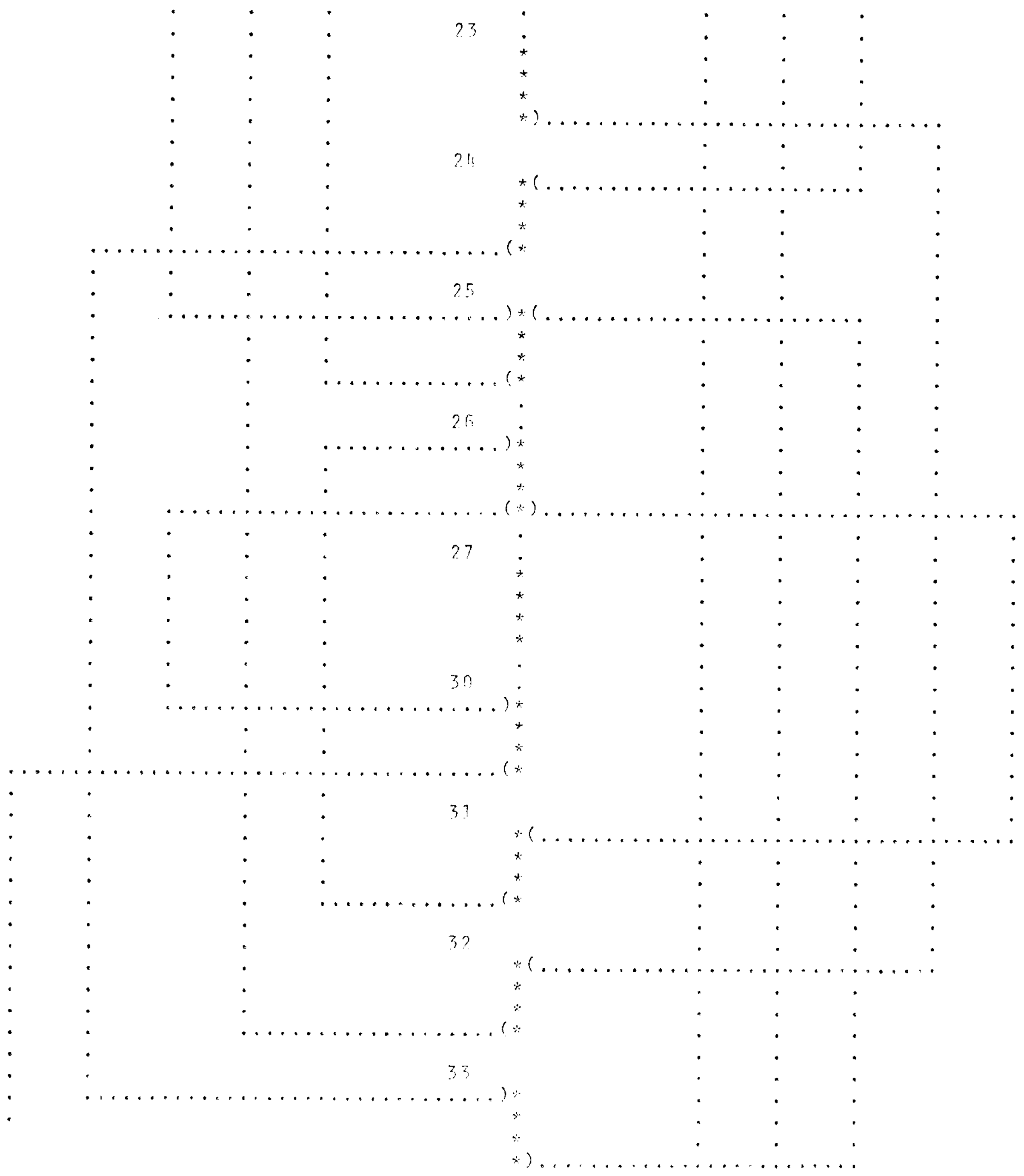
```
            ENTRY     DSRT18       SUBROUTINE TO SORT A LIST OF NUMBERS
*                                  THE SMALLEST NUMBER IS AT LOW OCTAL
*                                  THE CALLING SEQUENCE
*                                     TSX $SSORT18,4
*                                     PZE LOC OF HIGH OCTAL + 1 ADDRESS
*                                     PZE COUNT ADDRESS
DSRT18      SXA       IX1,1        SAVE IX1
            SXA       IX2,2        SAVE IX2
            SXA       SRTN,4       SAVE THE RETURN ADDRESS
            CLA*      1,4          GET HIGH OCTAL +1 ADDRESS
            SUB       A02          FORM HIGH OCTAL ADDRESS
            PAC       ,1           FORM 2'S COMP (HIGH OCTAL)
            TXI       *+1,1,2      FORM 2'S COMP (HIGH OCTAL) + 1
            SXA       SORTA,1      STORE IN SORTA
            CLS*      2,4          GET -COUNT
            TZE       SRTN         IF COUNT ZERO, RETURN
            ADM*      1,4          FORM HIGH OCTAL +1 -COUNT
            SUB       A02          FORM HIGH OCTAL - COUNT
            PAC       ,2           FORM 2'S COMP (HIGH OCTAL-COUNT)
            SXA       SORTB,2      STORE IN SORTB
            AXT       18,1
SORT1       LXA       SORTB,4      INITIALIZE SEPARATION ROUTINESET IX4=TOP OF
            CLA       SORTA+18,1
            PAX       ,2           SET IX2=BOTTON OF STBL
            CAL       SORTBT+18,1            PICKUP BIT TO BE SORTED ON
            TXI       SORT11,2,-2
SORT12      SXD       *+1,4        SCAN UP FOR ZERO-BIT
            TXH       SORT13,2,**            STOP AT TOP OR ON LAST SORTED SYMBOL
            LDI       2,2
            TIF       *+2          TRA ON FIRST ZERO BIT
            TXI       *-3,2,2
            LDQ       0,4
            STI       0,4
            STQ       2,2
            LDQ       1,4
            LDI       3,2
            STQ       3,2
            STI       1,4
SORT11      SXD       *+1,2        SCAN DONE FOR ONE-BIT
            TXL       SORT13,4,**            STOP AT BOTTOM OR ON LAST SORTED SYM
            LDI       2,4
            TIO       *+2          TRA ON FIRST SYMBOL W/ PROPER BIT
            TXI       *-3,4,-2
            TXI       *+1,4,-2
            TXI       SORT12,2,2
SORT13      TXI       *+1,2,2
            PXA       0,2
SORT3       STA       SORTA+19,1             SAVE LOC. OF LAST SORTED SYMBOL
```

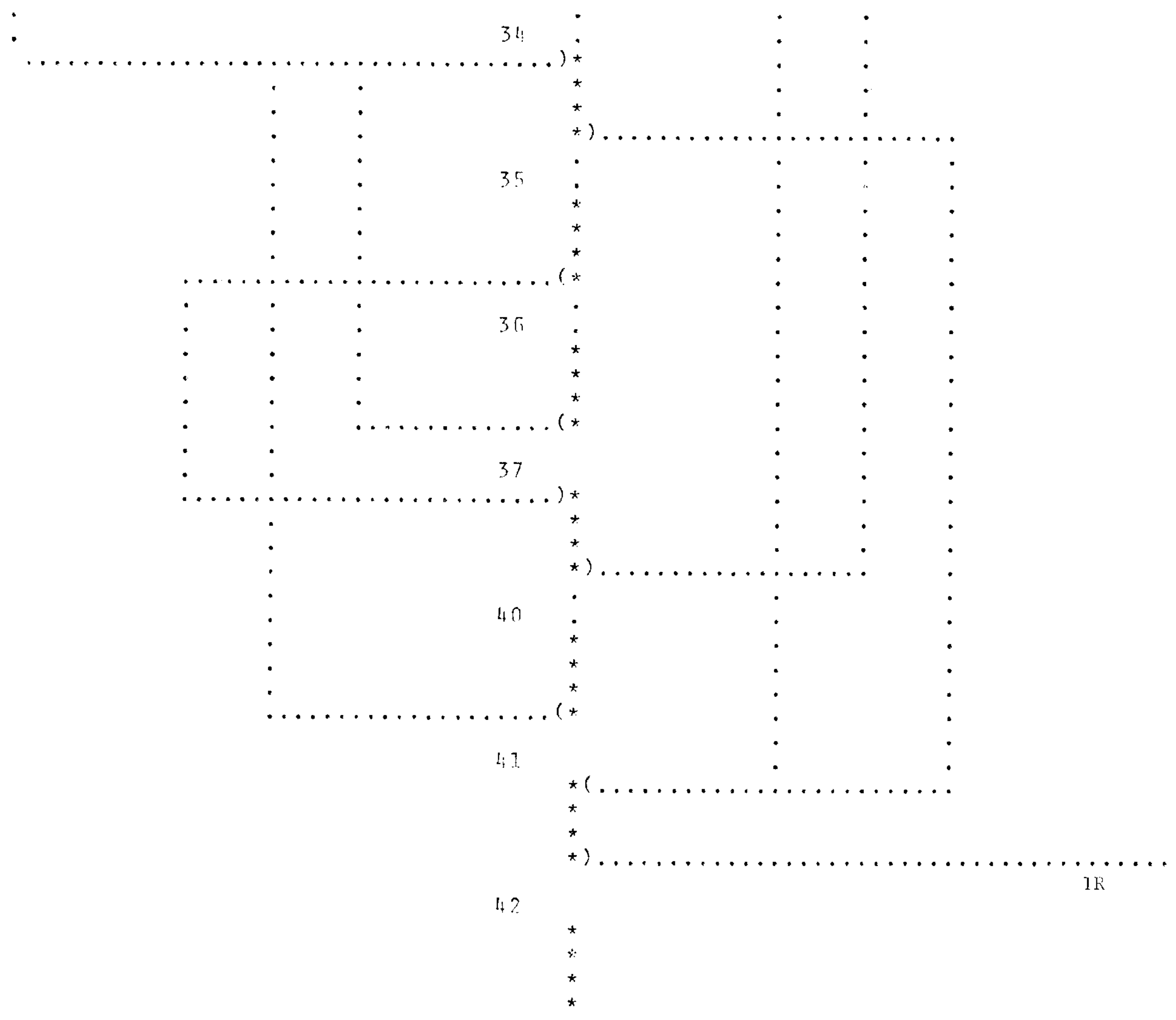
```
          CAS       SORTB     CAS W/ TOP OR W/ PREVIOUS LOC.
          TRA       SORT31    IF GREATER PICK UP EARLIER LOC.
          TRA       SORT4     IF EQUAL, SEE IF DONE
SORT32    TIX       SORT1,1,1 IF LESS, GO ON TO NEXT BIT
SORT87    AXT       1,1
          CLA       SORTA+18
          SUB       AQ2
          STA       SORTB
          CLA       SORTA+17
          STA       SORTA+18
          CAS       SORTB
          STA       SORTB
          TRA       SORT7
          TRA       SORT87
SORT31    CLA       SORTA+18,1          IF R2 IS ABOVE (SORTB),
          TRA       SORT3     SET IT TO BOTTOM OF BLOCK
SORT4     SUB       AQ2
          STA       SORTB     BRING SORTB UP TO DATE
          LDQ       SORTA+18,1          LOOK AT LAST LOC. EXCHANGED
          STQ       SORTA+19,1          MOVE IT UP
          TLQ       SORT32    CAS W/SORTB
SORT7     TXH       SRTN,1,17
          LDQ       SORTA+17,1          SEARCH BACK UP THROUGH THE TABLE
          STQ       SORTA+18,1
          TLQ       *+2       CAS TO SORTB AGAIN
          TXI       SORT7,1,1 GO AROUND AGAIN
          SUB       AQ2
          STA       SORTB     UPDATE SORTB AGAIN
          TLQ       SORT1     GO BACK FOR ANOTHER PASS
          TXI       SORT7,1,1 INCREMENT IX1 TO SEARCH TABLE
SRTN      AXT       **,4
IX1       AXT       **,1      RESTORE IX1
IX2       AXT       **,2      RESTORE IX2
          TRA       3,4
SORT77    PZE
SORTB     PZE
SORTA     DUP       1,19
          PZE
AQ2       PZE       2
SORTBT    OCT       400000,200000,100000
          OCT       40000,20000,10000,4000,2000,1000
          OCT       400,200,100,40,20,10,4,2,1
          END
```

Figure A2.3b - The Topological Flowchart for Figure A2.3a




12
13
14
15
16
17
20
21
22


23
24
25
26
27
30
31
32
33


34
35
36
37
40
41
1R
42

BIBLIOGRAPHY

1. Berge, C., The Theory of Graphs and Its Application, Translated by Alison Doig, John Wiley and Sons, New York, 1962.

2. The Compatible Time-Sharing System, A Programmer's Guide, M.I.T. Press, Cambridge, Massachusetts, 1966.

3. Haibt, L. M., "A Program to Draw Multilevel Flow Charts," Proceedings of the Western Joint Computer Conference, pp. 131-137, (1959).

4. Hain, G. and K. Hain, "Automatic Flow Chart Design," Proceedings of the ACM 20th National Conference, pp. 513-521, (August 1965).

5. IBM 7094 Data Processing System Reference Manual, IBM Corporation, Poughkeepsie, New York.

6. IBM 7090/7094 Programming Systems FORTRAN II Assembly Program (FAP) Reference Manual, IBM Corporation, Poughkeepsie, New York.

7. Iverson, K. E., A Programming Language, John Wiley and Sons, New York, 1962.

8. Krider, L., "A Flow Analysis Algorithm," Journal of the Association for Computing Machinery, Vol. 11, No. 4, pp. 429-436, (October 1964).

9. Needleman, M. R. and C. A. Irvine, Pathfinder, A Source Code Analysis Program for the Multiprocessor Environment, Western Data Processing Center, University of California, Los Angeles, California, (1966).

10. Nievergelt, J., "On the Automatic Simplification of Computer Programs," Communications of the ACM, Vol. 8, No. 6, pp. 366-370, (June 1965).

11. Prosser, R. T., Applications of Boolean Matrices to Analysis of Flow Diagrams, Technical Report No. 217, Lincoln Laboratory, Lexington, Massachusetts, (January 1960).

12. Ramamoorthy, C. V., "Connectivity Consideration of Graphs Representing Discrete Sequential Systems," IEEE Transactions on Electronic Computers, Vol. EC-14, No. 5, pp. 724-727, (October 1965).

13. Rising, H. K., On an Automated Method of Symbolically Analyzing Times of Computer Programs, Technical Report No. 154, MITRE Corporation, Bedford, Massachusetts, (March 1966).

14. Ross, D. T., "A Generalized Technique for Symbol Manipulation and Numerical Calculation," Communications of the ACM, Vol. 4, No. 3, pp. 147-150, (March 1961).

15 Sutherland, I. E., Sketchpad: A Man-Machine Graphical Communication System, Technical Report No. 296, Lincoln Laboratory, Lexington, Massachusetts, (January 1963).

BIOGRAPHY

Daniel U. Wilde was born on December 27, 1937 in Wilmington, Ohio. He attended high school at the Evanston Township High School, Evanston, Illinois, from which he was graduated in June, 1956. He was an undergraduate at the University of Illinois where he received the degree of Bachelor of Science from the Department of Electrical Engineering in February, 1961. Since then he has been a graduate student at the Massachusetts Institute of Technology where he received the degree of Master of Science from the Department of Electrical Engineering in September, 1962. Mr. Wilde is a member of Tau Beta Pi, Eta Kappa Nu, and Sigma Xi. Since 1962 he has been a member of the staff of the Electrical Engineering Department, first as a teaching assistant and then as a research assistant with Project MAC. Since 1964 he has been a Research Instructor of Medicine at the Boston University Medical School. He is married to the former Marylin R. Corbett of Billings, Montana.

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Massachusetts Institute of Technology Project MAC	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED
	2b. GROUP None
3. REPORT TITLE Program Analysis by Digital Computer	
4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)* PhD. Thesis, Electrical Engineering, June 1966	
5. AUTHOR(S) *(Last name, first name, initial)* Wilde, Daniel U.	

6. REPORT DATE August 1967	7a. TOTAL NO. OF PAGES 192	7b. NO. OF REFS 15
8a. CONTRACT OR GRANT NO. Office of Naval Research, Nonr-4102(01)	9a. ORIGINATOR'S REPORT NUMBER(S) MAC-TR-43 (THESIS)	
b. PROJECT NO. NR 048-189		
c. RR 003-09-01	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*	
d.		

10. AVAILABILITY/LIMITATION NOTICES

Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES None	12. SPONSORING MILITARY ACTIVITY Advanced Research Projects Agency 3D-200 Pentagon Washington, D. C. 20301

13. ABSTRACT Comparing properties of non- and self-modifying programs leads to the definition of independent and dependent instructions. Non-modifying programs contain only independent instructions, and such programs can be analyzed by a straight-forward, two-step analysis procedure. First, program control flow is detected; second, that control flow is used to determine program data flow or data processing. However, self-modifying programs can also contain dependent instructions, and then program control flows and data flows exhibit cyclic interaction. This cyclic interaction suggests using an iterative or relaxation analysis technique. Initially, the relaxation procedure determines a first approximation to control flow; the second step, to data flow. These two steps are repeated until steady-state condition is reached.

Algorithms for implementing the first iteration are presented. These algorithms are capable of analyzing programs which modify their control and processing instructions while executing. Also described are data structures which permit constructing functional expressions for data flow or information processing. Finally, actual output flowcharts of self-modifying programs are displayed.

14. KEY WORDS

Automatic flowcharting	Multiple-access computers	Real-time computers
Computers	On-line computers	Time-sharing
Machine-aided cognition	Program analysis	Time-shared computers

UNCLASSIFIED
Security Classification