THE UNIVERSITY OF ALBERTA

STRATEGIES FOR MICROPROGRAM OPTIMIZATION

© BY

WESLEY GARY SITTON

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

SPRING, 1973

ABSTRACT

The purpose of this study is to develop microprogram optimization strategies which include an analysis of alternate action forms in the deletion of nonessential actions. Alternate actions (micro-operations) are determined by the identification of all possible inputs and outputs for each action in a program node. The examination of alternate action forms provides a significant increase in the opportunities for nonessential action deletion since the program node may be restructured to force actions to assume a nonessential status. Many action form combinations may be used to generate different program node structures. It is necessary to identify a program node structure which results in the greatest number of action deletions. This optimal structure is not necessarily unique.

The study poses and solves three major problems: The identification of alternate and equivalent action forms; the definition of deletion strategies which include an analysis of alternate and equivalent action forms; and, the determination of a set of deletion strategies which may be employed in the optimal deletion of nonessential actions.

The deletion of nonessential actions is directly applicable to both Microprogram Optimization and General Compiler Optimization. Therefore, the results of this study may be shared by both disciplines.

ACKNOWLEDGEMENTS

The author wishes to extend special thanks to his thesis supervisor, Dr. J. Tartar for his invaluable guidance and assistance in this study.

Thanks go to Prof. W. S. Adams for his assistance in the early stages of this research.

The author also acknowledges Dr. T. A. Marsland and Dr. B. J. Mailloux for their time and assistance as members of the thesis committee.

Financial support was generously given by the Department of Computing Science, the National Research Council, and a University of Alberta Fellowship.

Finally, the author expresses great appreciation for the support and encouragement of his wife, Judy, whose many hours of proofreading have made the preparation of this thesis much easier.

CONTENTS

CHAPTER III (CONTINUED) PAGE

CHAPTER IV (CONTINUED) PAGE

CHAPTER V (CONTINUED) PAGE

LIST OF TABLES

LIST OF FIGURES

FIGURE PAGE

CHAPTER I

INTRODUCTION

The development of high-level microprogramming languages and their compilers has led to microprogram optimization research. Microprogram storage cost and frequency of execution are such that optimization is of critical importance.

Notable success has been attained in microprogram optimization by augmenting existing software compiler techniques, with ones that reflect the differences between software and firmware (micro-code); however, the basis of optimization remains the same for both: The identification and deletion of nonessential actions. Firmware actions are micro-operations which consist of one or more input memory sub-units, a primitive operation, and an output memory sub-unit.

Nonessential actions (micro-operations) are divided into three general classes: Negated, redundant, and parallel. The purpose of this study is to develop optimization strategies which include an analysis of alternate input and output specifications for each action. Alternate and equivalent action forms are to be employed to identify all possible nonessential actions. Previous deletion strategies have examined only the source form of each action; i.e., examination is given only to input and output memory sub-units specified at the source-level of each action. The deletion of all nonessential actions requires the identification and analysis of all possible forms of each action.

A deterministic algorithm is developed for the identification of all possible alternate and equivalent action forms; and then, optimization strategies are defined which include an analysis for all possible forms of each action. With analysis given to all possible forms, the definition of nonessential actions is expanded to be inclusive of the following actions.

1. Redundant actions: Those actions whose outputs are predictable and currently available from other equivalent action execution. Equivalent actions are those actions whose outputs are identical with respect to value.

2. Parallel actions: Those actions whose outputs may be defined by other equivalent actions.

3. Negated actions: Those actions whose outputs need never be used.

The approach used in this study is to identify alternate action forms by the analysis of equivalently defined memory sub-units. Deletion strategies are based upon an examination of alternate action forms which may be employed to force actions to conform to a nonessential status. Limitations of the approach are that run-time statistics are not used in the determination of equivalent memory sub-units; analysis is performed only upon program nodes, not regions; and, branch-point analysis is not performed.

The literature survey, Chapter 2, presents a review of microprogram optimization techniques as well as a general review of microprogramming. The general review of microprogramming includes a brief history, philosophies, and languages.

There exists a wide variety of microprogrammable machines and microprogramming languages; therefore, this study requires the proposal of a machine description and microprogram format which is sufficiently universal to avoid restrictive conclusions. Chapter 3 presents the machine description and microprogram format along with a detailed discussion of the study framework.

A major problem dealt with in the study is the identification of all alternate and equivalent action forms; i.e., given N sequential actions, all possible inputs and outputs must be identified. The solution to the problem is given in Chapter 4 in the form of an algorithm and proof of its correctness.

When alternate and equivalent action forms are determined, it is possible to force actions to assume a nonessential status. This is accomplished by the assignment of alternate input and output memory sub-units. A minimal set of conditions are met, in order for an action to assume a nonessential status. These conditions, defined in Chapter 5, are based upon a analysis of feasible alternate action forms and are applied to actions independently.

From the conditions for deletion candidacy, it can be seen that there may exist more than one strategy for the deletion of an action. Furthermore, the selection of a particular strategy is a dependent event, and may effect the deletion candidacy of other actions; therefore, it is necessary to define all possible deletion strategies and all conflicting deletion strategies. Chapter 6 defines the strategies which may exist for the deletion of an action; the conflicting strategies; and, the criteria of optimality in the selection of strategies.

Conclusions and recommendations for further research are offered in Chapter 7. Findings of the study include the following: The set of redundant actions is covered by the set of parallel actions; the algorithm which identifies alternate action forms also identifies all cases where parallel hardware fan-out may be exploited; an examination of adjacent memory sub-unit definitions is sufficient to identify all negated definitions; and, order-isomorphisms may exist in the analysis of deletion candidates, but are identifiable and avoidable.

The deletion of nonessential actions is directly applicable to both Microprogram Optimization and General Compiler Optimization. Therefore, the results of this study may be shared by both disciplines. In this study, optimization strategies are developed for microprograms, rather than software programs, for the following reasons.

1. Generally, micro-routines have a high frequency of execution.

2. Microprogram stores (control stores) are more expensive than software program stores (main memories).

3. Without loss of generality, subroutine calls and indirect addressing may be omitted from the study.

CHAPTER II

LITERATURE SURVEY

2.1 Introduction

This literature survey is designed to present a brief historical review of microprogramming along with a discussion of related research. In Chapter 3, a microprogram format is presented which is intended to include many different microprogramming languages and philosophies. In Section 2.3, discussion is given to different microprogramming philosophies, and Section 2.4 provides a description of language categories. Section 2.5 is devoted to a review of recent microprogram compiler optimization research.

2.2 Brief History of Microprogramming

Since its definition by M. V. Wilkes in 1951 [53], microprogramming has enjoyed increased popularity as a viable alternative to hard-wired computer control. Microprogramming research has generally been directed toward two major areas: application and language development. Examples of applications would include the development of virtual machine emulators such as a virtual APL machine [1] and a virtual FORTRAN machine [38]. Examples of language developments are presented in Section 2.4.

Microprogramming was developed to provide a systematic and orderly approach to the design of control systems for digital computers. The control system is described in terms of register transfers within the processor; transfers may take place through an adder or other logic circuitry.

The microprogram consists of a series of steps called micro-instructions which reside in a control storage device. As explained by Wilkes, the control storage device is made up of two READ-ONLY diode matrices, matrix A and matrix B. The A matrix routes the logic to be performed by the micro-instruction, and the B matrix is used to select the next micro-instruction for execution. A timing pulse along with a micro-instruction enters a decoding tree, and the appropriate connections are selected within matrix A. The B matrix output is then routed, via a delay circuit, to the micro-instruction address register; thus, the next micro-instruction is selected. Figure 2.1 displays the original model as defined by Wilkes.

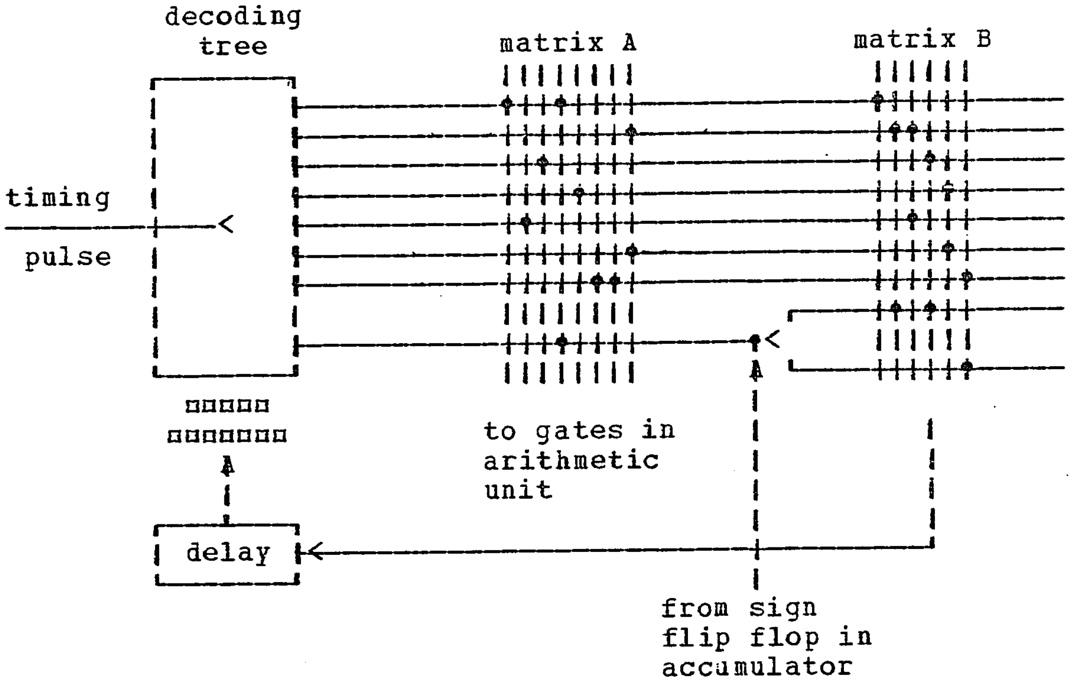

decoding tree
matrix A
matrix B
timing
pulse
delay
to gates in arithmetic unit
from sign flip flop in accumulator

Figure 2.1 Wilkes' Original Microprogramming Model[52]

The use of microprogramming exclusively as a design tool continued until the mid 1960's. Computers began to emerge which were designed with microprogramming capacities above the normal requirements of the host machine's instruction set. With the additional control storage capacity, the concept of using microprograms to support software became feasible. The term "firmware" was coined by Opler[43] in 1967 to describe this interaction of software and hardware.

User interest in microprogramming gained additional stimulation with the use of READ-WRITE control memories[6,24]. Writable control stores allow easy storage and alteration of microprograms. With the expanded microprogram capacity and writable control storage, both vendor and user became interested in easier ways in which to express microprograms. This led to the development of many different microprogramming languages.

Several languages and their compilers have been developed to facilitate the expression of microprograms. Languages vary greatly in both their objective and structure, with some languages expressed in an assembler-like format [55], while others assume a compiled macro expansion structure [13]. Two major reasons for the wide variety of language types are the variety of applications and the variety of microprogramming philosophies.

2.3 Microprogramming Philosophies

A study performed by Redfield[47] on the different techniques of microprogramming centered around three major microprogramming philosophies: monophase versus polyphase, parallel versus serial, and encoded versus highly encoded.

2.3.1 Monophase Versus Polyphase Microprogramming

Monophase and polyphase microprogramming (also called vertical and horizontal) are terms which refer to the measure of time for which a micro-instruction is active during the execution of a particular micro-routine. A micro-routine is defined as the set of micro-instructions required to implement a specific operation, such as a "branch and link" (BAL) machine-language instruction. In a monophase environment, the micro-instruction is active only for the clock cycle in which it is initiated; each micro-instruction causes a single simultaneous issue of control signals. In a polyphase microprogram, the micro-instruction generates control signals which are active over more than one clock cycle.

A strictly monophase approach may require impractical numbers of micro-instructions; thus causing excessive control storage fetches. However, a strictly polyphase approach may require an excessively large micro-instruction word.

2.3.2 Parallel Versus Serial Microprogramming

Parallel and serial microprogramming are terms which refer to the scheme used to select the next micro-instruction. The parallel approach utilizes a "best guess" philosophy by the selection of the next micro-instruction during the execution of the current micro-instruction. The selection may be made in a sequential manner, or it may be based upon data available from the micro-instruction which immediately preceded the currently active micro-instruction. In a serial approach, the next micro-instruction is selected upon the completion of the one currently being executed.

2.3.3 Encoded Versus Highly Encoded Microprogramming

Micro-instruction word formats lie on a continuum scale between direct control (minimally encoded) and highly encoded control. With the direct control scheme, each micro-instruction contains fields which control many different data paths within the processor unit. For example, a direct control micro-instruction, such as that proposed by Tucker and Flynn[51], may contain fields which control micro-instruction sequencing, masking, shifting, and adder operations (micro-operations). Computers which currently use a direct control micro-instruction word format include the RCA Spectra 70 and the IBM System/360 models 20, 25, 30, 40, 50, 65, and 85.

A highly encoded micro-instruction word format is analogous to the format of a one or two address machine-language instruction. The micro-instruction contains an operation code, which is decoded, and one or more operands which participate in the micro-operation execution. Two machines which use the highly encoded scheme are the Honeywell H-4200 and the Standard Computer IC-7000E.

In viewing each end of the continuous scale, it can be seen that the direct control approach requires a larger control storage word and is designed to exploit parallel data path operations; while the highly encoded approach requires more instructions and is generally more flexible with respect to algorithm expression. A further analysis of the relative efficiencies of these two formats is presented in separate articles by Jakolat[25] and Kurpanek[29].

The development of symbolic microprogramming languages paralleled the use of microprograms to augment software routines. Along with this development, interest has been generated in different micro-instruction word formats. The next section describes the two major categories of symbolic microprogramming languages.

2.4 Language Categories

Many languages for the expression of microprograms have been developed since 1967: Some draw heavily upon languages which have been developed expressly for the description of machine logic, such as Schlaeppi's LOTIS [49]. Microprogramming languages may generally be classified into two broad categories: Those which assume a particular hardware configuration for the host machine; and, those which include facilities for defining the host machine.

2.4.1 IBM's CAS Language

IBM's Control Automation System (CAS), designed for the System/360 line, is an example of the first category [24]. Microprograms are written as a series of micro-instruction boxes which specify the action to be taken by each micro-instruction. These boxes are connected in a flow chart manner to describe the microprogram. The information in each box is translated into a micro-instruction; then, the micro-instructions are converted, after simulation, into control storage bit patterns.

The format and content of micro-instructions change when microprogramming different models within the System/360 line; however, each micro-instruction box generally contains fields for arithmetic statements, local and main storage addressing, data transfer specifications, and control storage addressing. The micro-instruction box for the System/360 model 40 is given in Figure 2.2.

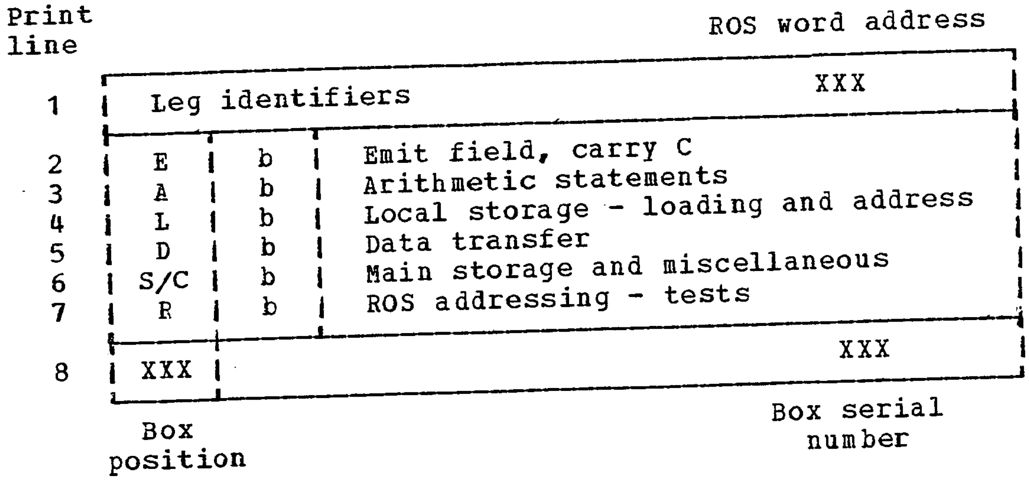

Print line
ROS word address
1
Leg identifiers
XXX
2 E b Emit field, carry C
3 A b Arithmetic statements
4 L b Local storage - loading and address
5 D b Data transfer
6 S/C b Main storage and miscellaneous
7 R b ROS addressing - tests
8 XXX
XXX
Box position
Box serial number

Figure 2.2 System/360 Model 40 Micro-instruction Box

The restrictive nature of machine-dependent microprogramming languages provides the primary stimulus for the development of languages which include facilities for machine description. The following section describes a language which has machine description capabilities.

2.4.2 Young's Language

A microprogram simulator developed by Steven Young of RCA is an example from the second language category [55]. The simulator includes facilities for the definition of machine elements, micro-operations, and micro-instructions. The system, ALSIM, is designed to allow machine definition and microprogramming of a wide range of machine configurations.

ALSIM is a highly formatted, descriptive language. Generally, the system is divided into the following three mutually dependent sections.

1. Machine description section: Defines the machine in terms of registers, busses, fields, functional memory, main memory, and processor functions.

2. Micro-operation section: Defines the micro-instruction format, the order of execution for functions within the micro-instruction, and hardwired or emulated primitive functions.

3. Micro-instruction section: Consists of micro-instructions coded in the format defined in the micro-operation section.

2.4.2.1 Machine Description Section

The definition of machine elements is accomplished through a specification of storage registers, data paths, memories, and basic machine functions. The general form of a machine element definition is

```
        DEFINE name, key word, qualifiers
```

where "key word" specifies the type of machine element to be defined. For example, the following machine element definitions declare a 32-bit adder (ADD), register (A), and bus (TOT).

```
        DEFINE A,REG,32
        DEFINE TOT,BUS,32
        DEFINE ADD,ADDER,32,0,OV,31,C,CS,0
```

Qualification for the adder includes the specification of an overflow register and bit, along with carry registers.

2.4.2.2 Micro-operation Section

The micro-operation section is used to define the micro-instruction format, execution order of micro-instruction functions, test conditions, and primitive functions. The micro-instruction format is specified by use of the MAP statement. For example,

```
        MAP A,T,F,F,F,G,G
```

specifies that the format of the micro-instruction is to be first, current address(A); next, a test condition(T); then, three functions(F); and lastly, two branch addresses(G).

The execution order of the functions in the micro-instruction is specified on the EOM statement. The EOM statement specifies when functions are to be performed; when tests are to be evaluated; and, which addresses are to be selected.

EQN statements are used to define tests and assign names to them. These names are used in the code specified in the micro-instruction section. For example,

```
EQN A:TOT ACOMPSUM
```

assigns the name ACOMPSUM to the comparison of A and TOT.

Primitive functions are defined by the use of the FUN statement. The FUN statement defines operations to be performed upon machine elements. The operations are specified in terms of basic logic functions such as AND, OR, and TR (transfer). To define a function which would transfer register A to bus TOT, the following FUN statement may be used.

```
FUN TRA TR,TOT,A
```

2.4.2.3 Micro-instruction Section

The micro-instruction section is used to write the micro-routines. Micro-instructions are coded in the format of the MAP statement; with the execution order as given on the EOM statement; the mnemonic test names as given on EQN statements; and, the functions as defined on the FUN statements.

In summary, the model provides an effective means for the description of a wide range of machine configurations. Furthermore, from the standpoint of assembler microprogramming, ALSIM is easy to use and follows a logical flow of machine declaration, micro-instruction format description, and micro-instruction action.

ALSIM and IBM's CAS have been presented to provide examples of different microprogramming language approaches. It can be seen that the two languages vary greatly in both their structure and programming requirements. However, the micro-instruction level of all microprogramming languages is functionally the same: Primitive hardware functions are invoked for the purpose of defining or changing the state of machine elements. It is to this micro-instruction level that optimization strategies for microprograms are developed in this study.

2.5 Microprogram Optimization

The development of microprogram optimization strategies has made extensive use of techniques developed for software compiler optimization. Many techniques described in a text by Gries[20] and developed by Allen[2] and Lowry and Medlock[35] have been applied to micro-code optimization. As in software optimization, the microprogram is first divided into regions and program nodes within regions; then, optimization techniques are applied to each program node.

2.5.1 Regional Representation of Microprogram

In a broad perspective, for global optimization the microprogram is represented as a directed graph whose regions are operated upon to eliminate nonessential operations, optimize branch points, and maximize the exploitation of parallel hardware units [7,27,45,52]. The microprogram is represented as a linear list, R(i), of regions, R(1),R(2),...,R(n), with each region made up of program nodes. A program node is a linear sequence of micro-operations which has one entry point (first micro-operation) and one exit point (last micro-operation). Branches (e.g., GO TO), if any, appear as the last actions in a node.

Nodes are connected by arcs. These arcs represent a branch and are used to identify successor and predecessor nodes as well as articulation nodes. An articulation node is any node in a strongly connected region which is assured of being executed, if the region is entered. Thus, an articulation node lies on every entry and exit path of a region (see Figure 2.3). Regions are constructed with the following properties.

1. R(i) is strongly connected: A strongly connected region is a set of nodes in which there is a flow path of arcs from any node in the set to any other node in the set.

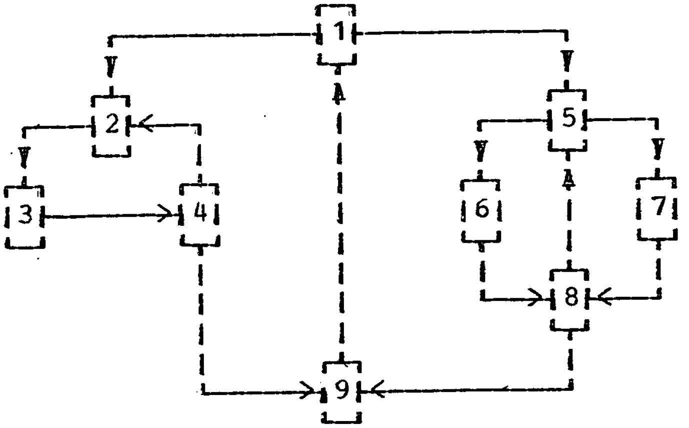
2. R(i) ≠ R(j).

3. For i < j

either $R(i) \cap R(j) = \emptyset$

or $R(i) \cap R(j) = R(i)$

i.e., R(i) is nested in R(j).



(a)

region	entry node	exit node	articulation nodes	region nodes
R(1)	5	8	5,8	5,6,7,8
R(2)	2	4	2,3,4	2,3,4
R(3)	1	9	1,9	1,2,3,4,5 6,7,8,9

(b)

Figure 2:3 Regional Representation of Program Nodes: (a) Program Directed Graph; (b) Region Breakdown.

Optimization of regions is performed on the following basis:

1. When R(i) is being optimized R(1),R(2),...,R(i-1) have already been optimized.

2. When R(i) covers other regions, only nodes not contained in those regions must be examined. The set, R'(i), of unexamined nodes in R(i) is determined by

$$R'(i) = R(i) \cap \left[\bigcap_{j=1}^{i-1} \overline{R}(j) \right] .$$

2.5.2 Optimization Techniques

Using a model defined by Cook and Flynn[11], Klier and Ramamoorthy[27] investigated the use of the following optimization techniques.

1. Identification and deletion of nonessential actions.

2. Static frequency analysis of busy variables.

3. Application of Tumasulo's algorithm for machine data-flow fan-out[52].

4. Optimization of branch-points by code rearrangement.

Nonessential operations (actions) which may be deleted are defined by Klier and Ramamoorthy as follows:

> "Redundant actions are those whose outputs are predictable and currently available from previous identical action executions. ...
> Negated actions are those activities whose results are never used."

Conditions for the deletion of a successor action as redundant are given as follows. (from [27])

1. Two actions have the same inputs, operation, and output specification.

2. No input sub-unit is redefined between the two identical actions.

3. The output sub-unit is preserved.

4. If one of the two actions is executed, then the other action is also executed.

These conditions substantiate Theorem 13 in [50] by outlining the rules by which a redundancy algorithm may be constructed. It is also interesting to note that the register instructions defined in [50] may be mapped onto the micro-instruction format presented in [27]; i.e., software loop register operations may be expressed in a microprogram notation.

Also in [27], the condition given for the deletion of an action as negated is that the output from an action appears as an output in a succeeding action, without first appearing as an input.

The important limitations in the above conditions are that only identical actions are considered, and equivalent sub-units are not identified; i.e., only formal identities are considered. Therefore, alternate forms of an action are not considered.

A review of techniques for the identification and exploitation of parallel operations is presented in [45]. Tumasulo's algorithm [52] is referenced by Klier and Ramamoorthy [27] as a means of implementing the deletion of actions which have the same output value. Developed for the IBM/360 model 91, the algorithm makes use of an operation stack and reservation stations to optimally queue variables which are to be fanned-out to parallel hardware units.

Software compiler optimization and microprogram optimization have dealt with only identical actions and identical input and output memory sub-units, in the determination of nonessential actions; i.e., only the form of the action given at the source-level is considered. Chapter 3 describes the framework for the study, whose major objective is the development of optimization strategies which include an analysis of alternate and equivalent action forms.

CHAPTER III

ANALYSIS FRAMEWORK AND PROBLEM DESCRIPTION

3.1 Introduction

The purpose of this Chapter is to describe the general framework for the problem analysis. Major sections include the following: The proposal of a machine description and microprogram format; a discussion of the problem analysis framework; and, the presentation of a graphical description of the problems addressed in this study.

The machine description and microprogram format are proposed with the objective of including features found in many current machines and microprogramming languages. The microprogram format is presented as a notation for the general description of a microprogram.

The general framework and limitations of the study are presented in the section dealing with its scope. First, the problems dealt with in the study are described; second, the range of optimization is defined; third, the information collected for the problem analysis is discussed; fourth, nonessential actions are formally defined; and fifth, limitations of the analysis are presented. The last major section in the Chapter is a general discussion of a graph-theoretic approach to the problems addressed in the study.

3.1.1 Note on Definition Formats

Definitions which are used throughout this thesis (global) are preceded by the symbol sdef or def. sdef is the definition of a symbol which is presented to minimize verbosity and has no intrinsic relationship to conceptual development. def is a definition required to formulate a concept, theorem, proof, etc.

In order to succinctly describe the problems and solutions given in this thesis, it was necessary to develop additional notation. Wherever possible, notation is consistent with the existing literature. However, because of the nature of the problems addressed in the study, extensions to existing notation were required.

3.2 Machine Description and Microprogram Format

This section presents a general host machine description along with a description of a microprogram format. The microprogram format is closely aligned with the format described in [27] and is intended as a tool for the representation of a wide variety of microprogramming languages.

3.2.1 Machine Description

To avoid a machine-dependent presentation, the host machine is described in general terms so as to include the aggregate features of many present-day computers. The host is the machine which contains those functional hardware elements necessary to decode and directly execute each micro-operation used by the microprogram. A micro-operation is an operation which invokes a primitive hardware function (e.g., ADD, GATE, etc.), and requires a specific quantum of time to be executed.

The host machine is assumed to have the following hardware elements.

1. Operational units: Units which are invoked during the execution of primitive operations. Examples of operational units are arithmetic units, selector gates, and Boolean logic units.

2. Control Storage: The memory device which holds the microprogram. This device may be READ-ONLY or WRITE-controllable storage [24].

3. Main Memory: The low-speed memory device from which machine-language instructions are fetched to be decoded, and ultimately executed by microprograms. The main memory may also contain data which is referenced by the machine-language program.

4. Scratch-pad Memory: This is considered as high-speed memory (operating at approximately internal clock speed). It is used as a local store for an internal register file; intermediate result store for microprograms; system

accounting storage; or, for any other application which requires fast storage [17].

5. Data Paths: These are the paths which connect the memory units to the operational units. The network of data paths together with the operational units and memory units succinctly describe the host machine.

The host machine is further qualified by the following:

NP Number of primitive operations.

NS Number of memory cells (also called memory sub-units) which may be used as input and/or output to primitive operation i, $1 \leq i \leq NP$. Note: Generally, these memory cells reside in high-speed memory.

NO Number of operational units.

T A two-dimensional array which represents the internal clock time, $T(i,j)$, to execute primitive operation i, using operational unit j, $1 \leq i \leq NP$, $1 \leq j \leq NO$.

P A two-dimensional array which represents the possible participation, $P(i,k)$, of memory sub-unit k as the input and/or output to primitive operation i, $1 \leq k \leq NS$, $1 \leq i \leq NP$. $P(i,k) = 1$ if participation may occur, 0 otherwise.

These qualifications will aid in the discussion of the microprogram format presented in the next section.

3.2.2 Microprogram Format

A notation for the general representation of microprograms is presented below. For the purpose of this study, it is assumed that the compiler and/or assembler has generated the microprogram in the form described below. No attempt is made here to describe this translation process; it is, however, asserted that such a translation is feasible in terms of current technology.

To avoid a restrictive format, the microprogram is described to include highly encoded (vertical) and direct (horizontal) microprogram formats. It is noted that the difference between the two microprogram types is on a continuum scale; and, such a universal definition likely lacks the direct inclusion of technical characteristics which are unique to a specific language and machine configuration (e.g., emit fields, masks, etc.). This does not detract from the generality of results, as provisions are made for such technical qualifications of micro-instructions.

The following definitions are given before the discussion of the microprogram format is presented.

def. 3.1 Source Unit: The memory sub-unit used as input to a micro-operation.

def. 3.2 Sink Unit: The memory sub-unit to which the result of a micro-operation is gated.

def. 3.3 Microstep: Time required to execute one micro-instruction.

def. 3.4 Micro-instruction: The unit of a microprogram which contains the micro-operation(s) to be executed in one microstep.

def. 3.5 Micro-operation: The unit of a micro-instruction which causes a primitive hardware function to be invoked. Generally, a micro-operation specifies operational units, source unit(s), and sink unit(s). The micro-operation is functionally defined by the following operational classes.

a) Data-flow actions: Those micro-operations which cause a transfer of data within the machine.

b) Content permutation actions: Those micro-operations which define or change the state of memory sub-units.

c) Directional control actions: Those micro-operations which control the selection of the next micro-instruction (and micro-operation) to be executed.

It is important to note that the terms micro-operation and action are synonymous; however, the latter is used where functional qualification serves to clarify a point.

The definitions and qualifications given previously have established a basis upon which a general microprogram notation may be developed. The notation presented below is similar to that described by Klier and Ramamoorthy [27]. The {c}, and {TV} qualifications have been added to the notation given in [27], in order to include timing constraints and control information. Also, because this study deals with alternate memory sub-unit assignments, it will be necessary to reference the memory sub-units by subscript notation.

Given that there are H micro-instructions and that there may be L micro-operations defined by each micro-instruction, the following notation is presented to describe a microprogram.

$$\left[I(p,q): \quad Op \quad \{sc\} \quad \{sk\} \quad \{c\} \quad \{OU\} \quad \{TV\} \right]_{q=1,2,\ldots,L}^{p=1,2,\ldots,H}$$

where

I An index which uniquely identifies the micro-operation within the micro-instruction.

Op One of the primitive operations.

{sc} The memory sub-unit(s) used as the source for Op.

{sk} The memory sub-unit(s) used as the sink for Op.

{c} A control field which contains information such as control bits, shift amounts, selection masks, etc.

{OU} The operational unit(s) required to execute Op.

{TV} Time validity, T(i,j), in terms of internal clock cycles, of executing Op(i) using {sc}, {sk}, and {OU}, $1 \leq i \leq NP$, $1 \leq j \leq NO$.

As an example, consider a micro-instruction encoded as described by Tucker and Flynn[51]:

R3:=R5+R6, R7:=-2/R7, GO TO P2.

The micro-instruction is divided into three micro-operation fields: ADD, SHIFT, and SEQUENCE. It is coded to add R6 and R5 and then gate the result to R3, shift R7 left two bits, and select P2 as the next micro-instruction address (/R7 denotes contents of R7 are to be shifted). Assuming the micro-instruction was the second one in a program, its expression in the notation given would be as follows (assume address of P2 is held in R4). The {OU} and {TV} fields are not shown.

Index	Op	sc	sk	c
I(2,1):	ADD	{5;6}	{3}	
I(2,2):	SHIFT	{7}	{7}	{-2}
I(2,3):	JUMP	{4}		

The microprogram format has been presented as a means for the general expression of a wide range of microprogramming languages. It also provides a basis upon which the problem analysis framework may be described.

3.3 Problem Analysis Framework

The purpose of this section is to describe the framework of the study in terms of the following: The problems addressed in the study; the set of actions considered for optimization; the information collected from the set of actions; the actions which are considered for deletion; and, the limitations of the analysis.

3.3.1 Problem Description

Simply stated, the problem undertaken in this study is the deletion of nonessential actions from a set of sequentially executed micro-operations. Previous attempts at the deletion of nonessential actions have relied upon an evaluation of only the source form of each action; however, the deletion of all nonessential actions requires the identification and evaluation of all possible forms of each action. For example, consider the following example of four sequentially executed, highly encoded actions.

```
I(1):   ADD     {3;4}    {5}
I(2):   GATE     {5}     {6}
I(3):   GATE     {3}     {7}
I(4):   ADD     {7;4}    {5}
```

In this example, an examination of only the source-level form of each action yields no nonessential actions. An evaluation of the equivalent memory sub-units, {3} and {7}, yields another form for I(4), ADD {3;4} {5}. This alternate form for I(4) is redundant with respect to I(1) and is, therefore, nonessential.

The significance of the problem is based upon the premise that nonessential actions are most likely to occur in action forms which are alternate and equivalent to source forms. Nonessential action deletion is directly applicable to both Microprogram Optimization and General Compiler Optimization; therefore, the results of this study may be shared by both disciplines.

Optimization strategies are to be developed which include an analysis of all possible action forms. In order to achieve this goal, the following three major problems are addressed in this study.

1. All possible forms for each action must be determined. The variables of form for an action are input and output memory sub-units. Therefore, the problem is to determine all memory sub-units which may participate as inputs and/or outputs for each action.

2. Given that alternate action forms are determined, conditions for the deletion of an action must be defined in terms of alternate action form assignments. The problem is to define a minimal set of conditions which must be met in order for an action to be a negated, redundant, or parallel candidate for deletion.

3. Given a set of deletion candidates, all possible deletion strategies must be identified for each deletable action. The problem is to identify the strategies available for the deletion of an action; and, to specify conflicting strategies. With all strategies identified and conflicting strategies determined, the final problem is the determination of optimal deletion criteria.

In a broad sense, the solution to the above problems requires the following: The identification of alternate action forms; and, the development of strategies based upon deletion advantages gained through alternate action form assignments. A general statement of the approaches used in the solution of the problems described is presented next.

3.3.1.1 General Statement of Solution Approach

With N sequentially executed actions, i equivalent actions $(i \leq N)$, and j equivalently defined output units $(j \leq i)$, examination is given to determine all possible forms, F(m), of action(m), m=1,2,...,N. Equivalent actions and equivalent units are used to identify all possible source and sink unit combinations for each action; i.e., variables of form for actions are source and sink units.

An analysis of F(m) is performed to identify all actions which may be deleted. A minimal set of conditions is defined for negated, redundant, and parallel actions. Those actions which conform to at least one set of deletion conditions make up the set of actions, D, which are candidates for deletion.

The strategies which may be employed in the deletion of each action in D are defined. Also, the conflict of strategies is defined, since the strategies employed in the deletion of one action may affect the strategies available for the deletion of other actions. With strategies and strategy conflicts defined, it is possible to select one form, Z(k), for each action, which will result in the optimal deletion of actions from the original set of N actions, Z(k) $\in$ F(m), k=1,2,...,N.

The following section describes the range of actions over which the optimization strategies are to be applied; i.e., the set of N actions to be examined is described.

3.3.2 Optimization Range: The Program Node

Analysis of nonessential actions is performed only upon nodes in properly nested (or simply nested) regions of the microprogram [2,7]. The microprogram notation described is used in all program node examples. The definitions given below will aid in the discussion and analysis of program node examples. Throughout the remainder of this thesis, the term "unit" will generally be used in place of the more verbose "memory sub-unit".

sdef. 3.1 N: Number of actions in a program node.

<u>sdef</u>. 3.2 NS: Number of memory sub-units.

<u>sdef</u>. 3.3 nA: Number of elements in A, where A may be any defined variable.

<u>sdef</u>. 3.4 MO(i): The ith action in a program node, $1 \leq i \leq N$.

<u>sdef</u>. 3.5 OP(i), sk(i), and c(i): The primitive operation, sink unit, and control field respectively, specified in MO(i).

<u>sdef</u>. 3.6 sc(i,j,k): Reference source units within MO(i) and may have one (i), two (i,j), or three (i,j,k) subscripts.

sc(i) references all possible source unit combinations (forms) for MO(i); e.g., sc(4) references all possible source unit combinations for MO(4).

sc(i,j) references the jth combination of source units for MO(i), $1 \leq j \leq nsc(i)$; e.g., sc(2,3) references the 3rd source unit combination for MO(2). Note: j=1 always references the source unit combination given in the source form of the action.

sc(i,j,k) references the kth source unit in the jth source unit combination for MO(i), $1 \leq k \leq nsc(i,j)$; e.g., sc(4,3,1) references the 1st source unit in the 3rd source unit combination for MO(4).

The program node example given in Figure 3.1 is used as a basic example in the explanation of algorithms and analysis techniques. To reduce complexity, action examples do not contain the {OU} and {TV} fields given in the microprogram notation. It is assumed that before code changes are made, precautions are taken to insure against the creation of timing hazards.

To further reduce complexity, action examples will contain a maximum of two source units and one sink unit; also, nodes will contain only highly encoded microinstructions. These simplifications are not restrictive, as the theoretical framework of the thesis is based upon any encoding structure and arbitrary source unit bounds.

```
I(1):    GATE    {2}      {3}
I(2):    GATE    {4}      {6}
I(3):    ADD    {3;6}     {7}
I(4):    ADD    {1;2}     {2}
I(5):    ADD    {2;4}     {8}
I(6):    GATE    {7}      {5}
I(7):    ADD    {2;6}     {9}
I(8):    SUB    {7;3}     {3}
```

Figure 3.1 Program Node Example

It is useful to represent a program node in terms of the unit assignments to each action. This is done by the creation of a unit assignment table (UA) for each program node. The structure of the unit assignment table is given in the following section.

3.3.2.1 Unit Assignment Table (UA)

Given that there are NS memory sub-units (e.g., registers) and N actions represented in the node, the unit assignment table will be an (N+1) by NS array. The additional row, UA(N+1,j), is used to represent those units which are busy on exit from the node [35]; i.e., those units used as inputs in successor nodes, before they are redefined. UA(N+1,j) = 1 if unit {j} is busy on exit, 2 otherwise. UA(i,j) = 0|1|2|3 respectively, if unit {j} is not used, used as a source unit, used as a sink unit, or used as a source and sink unit for MO(i), i=1,2,...,N. (vertical bar, |, used throughout as an OR symbol)

For example, if units {3} and {8} are busy on exit from the program node given in Figure 3.1, then the unit assignment table generated for the program node is as given in Table 3.1. For Table 3.1, N = 8 and NS = 9.

Table 3.1 Unit Assignment Table for Program Node Given in Figure 3.1.

	Memory Sub-units								
Index	1	2	3	4	5	6	7	8	9
I(1)	0	1	2	0	0	0	0	0	0
I(2)	0	0	0	1	0	2	0	0	0
I(3)	0	0	1	0	0	1	2	0	0
I(4)	1	3	0	0	0	0	0	0	0
I(5)	0	1	0	1	0	0	0	2	0
I(6)	0	0	0	0	2	0	1	0	0
I(7)	0	1	0	0	0	1	0	0	2
I(8)	0	0	3	0	0	0	1	0	0
EXIT	2	2	1	2	2	2	2	1	2

The unit assignment information, along with equivalent action and equivalent unit information, is used in the determination of nonessential actions. The following section presents definitions of equivalence relations to be collected from the program node.

3.3.3 Information Collected from Program Node

As stated earlier, a major problem addressed in the study is the determination of alternate action forms. The variables of form for actions are source units and sink units. In order to determine all possible combinations of source units for each action, it is necessary to identify equivalent units. In order to determine all possible sink units for each action, it is necessary to identify equivalent actions. Simply stated, the information to be collected from the program node is that information required to identify alternate source and sink units for each action. The information to be collected is presented in the following definitions.

sdef. 3.7 QMSU: Memory sub-unit qualified by its index of definition. A QMSU is an ordered pair (u,uid), where u is the identifier of the unit, and uid is the index of the action which defines u. For example, (3,6) represents unit {3} as defined by the 6th action in the program node. If the unit is busy on entry into a program node, its index of definition is zero.

The qualification of units is an important feature of this study. It allows sequential analysis of actions to be performed without the traditional program optimization problems created by redefinition.

def. 3.6 Equivalent units: Those QMSUs whose contents are identical; i.e., where $1 \le i, m \le NS$ and $1 \le j, k \le N$,

$(i,j) <=> (m,k)$

if $MO(j) <=> MO(k)$.

def. 3.7 Equivalent actions: Those actions whose outputs are identical with respect to value; i.e., where $1 \leq i, j, k \leq N$,

$MO(i) <=> MO(j)$

if one of the following is true:

$Op(i) = Op(j)$, $c(i) = c(j)$, and $sc(i,1) <=> sc(j,1)$;

or, for $j < k$,

$Op(k) =$ data-flow, $sk(k,1) = i$, and $c(k) = \emptyset$;

or, for $j > k$,

$Op(j) =$ data-flow, $sk(j,1) = k$, and $c(j) = \emptyset$.

It is important to note that equivalent units and equivalent actions conform to transitive laws.

def. 3.8 Alternate source units: Those QMSUs which may be defined before $MO(i)$ and are equivalent to a QMSU specified in the source form of $MO(i)$; i.e., where $1 \leq m, t \leq NS$ and $1 \leq k, s, i \leq N$,

(m,k) is an alternate source unit for (t,s) at $MO(i)$

iff $k < i$ and $MO(k) <=> MO(s)$.

def. 3.9 Alternate sink units: Those QMSUs which are equivalently defined; i.e., where $1 \leq m, t \leq NS$ and $1 \leq k, s \leq N$,

(m,k) and (t,s) are alternate sink units for each other

iff $MO(k) <=> MO(s)$.

With the unit assignment table described and the equivalence relations defined, it is now possible to formally define all nonessential actions considered in this study. The definitions given in the next section qualify nonessential actions to include general deletion strategies.

3.3.4 Nonessential Action Definitions

Nonessential actions are defined in terms of the unit assignment table and equivalence relations. As given in Chapter 1, nonessential actions include three classes: Negated, redundant, and parallel. This section further qualifies the classes as parallel-forward, parallel-backward, redundant-forward, redundant-backward, and negated-forward. This qualification is required in order to represent all possible deletion strategies; e.g., if MO(i) and MO(j) are parallel actions, MO(i) <=> MO(j), then sk(i) may be moved to MO(j), or sk(j) may be moved to MO(i).

<u>def</u>. 3.10 Parallel-forward action: MO(i) is parallel-forward with respect to MO(j), $1 \le i,j \le N$, (and may be deleted) iff MO(i) <=> MO(j);

$i < j$;

$$\sum_{m=i+1}^{j-1} UA(m,sk(i)) = 0;$$

$UA(j,sk(i)) = 0|2$;

Either sk(i) may replace sk(j) at MO(j)

or sk(i) may be fanned-out at MO(j).

def. 3.11 Parallel-backward action: MO(j) is parallel-backward with respect to MO(i), $1 \leq i, j \leq N$, (and may be deleted)

iff MO(i) <=> MO(j);

$i < j$;

where $sk(i) \neq sk(j)$

$$\sum_{m=i+1}^{j} UA(m, sk(j)) = 2;$$

where $sk(i) = sk(j)$

for $m = i+1, i+2, \ldots, j-1$

$UA(m, sk(j)) \leq 1$;

Either sk(j) may replace sk(i) at MO(i)

or sk(j) may be fanned-out at MO(i).

def. 3.12 Redundant-forward action: MO(i) is redundant-forward with respect to MO(j), $1 \leq i, j \leq N$, (and may be deleted)

iff MO(i) <=> MO(j);

$i < j$;

$sk(i) = sk(j)$;

$$\sum_{m=i+1}^{j} UA(m, sk(i)) = 2.$$

def. 3.13 Redundant-backward action: MO(j) is redundant-backward with respect to MO(i), $1 \leq i, j \leq N$, (and may be deleted)

```
iff MO(i) <=> MO(j);
    i < j;
    sk(i) = sk(j);
    For m=i+1,i+2,...,j-1
        UA(m,sk(j)) ≤ 1.
```

Theorem 3.1 An evaluation of all parallel actions is sufficient to identify all redundant actions.

Proof:

By an examination of def 3.10, 3.12, 3.11, and 3.13, it can be seen that the conditions for forward and backward redundancy are included in the conditions for forward and backward parallelism. If MO(i) is redundant with respect to MO(j), then sk(i) = sk(j) and replacement is trivial; therefore, the conditions for parallelism, with sk(i) = sk(j), are identical to the conditions for redundancy. Hence, the set of all redundant actions in a program node is covered by the set of all parallel actions.

Q.E.D.

<u>def</u>. 3.14 Negated-forward action: MO(i) is negated-forward with respect to MO(j), $1 \le i,j \le N$, (and may be deleted)

iff $sk(i) = sk(j)$;

$i < j$;

$$\sum_{m=i+1}^{j} UA(m,sk(i)) = 2.$$

From Theorem 3.1, it can be seen that an evaluation of parallel-forward, parallel-backward, and negated-forward actions is sufficient to identify all nonessential actions. Previous research [27] has presented a separate analysis of parallel and redundant actions. Theorem 3.1 provides a more general basis of nonessential action analysis; since, nonessential actions may now be redefined as negated and parallel.

The general approach used in this study may now be stated in terms of these nonessential actions: It is to identify those alternate unit assignments which may be made in order to create a parallel-forward, parallel-backward, or negated-forward condition in the unit assignment table. The following section describes the specific limitations imposed upon the development of the approach.

3.3.5 <u>Limitations of Analysis</u>

The following are the limitations in the analysis of nonessential actions, as proposed in this study.

1. Analysis is performed only upon nodes; i.e., regional analysis is not included in the study.

2. No run-time statistics are used in the identification of equivalence relations. For example, under the analysis proposed in this study, QMSUs (3,1) and (2,3) are not identified as equivalent in the following program node.

```
I(1):    GATE     {6}      {3}
I(2):    ADD      {3,7}    {4}
I(3):    SUB      {4,7}    {2}     (assume {4} - {7})
```

Of course, if run-time statistics were collected, QMSUs (3,1) and (2,3) would be identified as equivalent.

3. Directional control actions are not considered as candidates for deletion.

The framework and limitations of the study have been discussed, and the major problems addressed in the study have been described. The next section presents a graph-theoretic description of the major problems. It is presented to provide a concise description of the optimization objective function and the combinatoric nature of the problems undertaken in this study.

3.4 Graph-Theoretic Description of Problem

The objective function is satisfied, if all nonessential unit definitions are deleted from the program node. Clearly, the problem involves combinatorics: To find the combination of units and primitive operations which will negate the largest number of micro-operations. The solution involves the identification and evaluation of all possible source and sink units. The problem may be expressed by the representation of the program node as a directed graph (similar to a dependency graph [2]). Sink units are connected to source units by operational arcs, with each action represented as a level in the tree. Unmodified ($c(i) = \emptyset$) data-flow actions are represented by vertical operational arcs. For example, the following program node is represented by the directed graph in Figure 3.2.

```
I(1):     GATE     {2}      {3}
I(2):     GATE     {4}      {6}
I(3):     ADD     {3;6}     {7}
I(4):     GATE     {7}      {5}
```

For the analysis of the above program node, assume that units {2} and {4} are defined on entry into the program node, and that the I(4) definition of unit {5} is used in successor nodes. Examination of the code shows that the

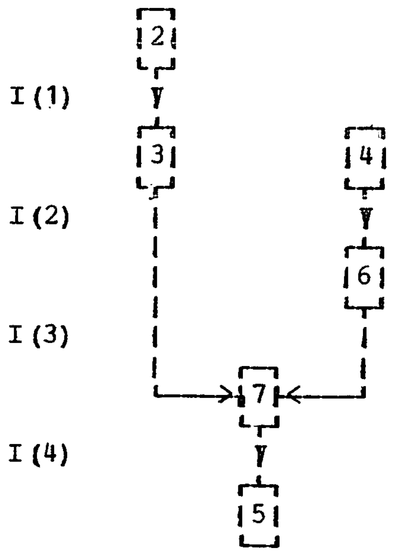
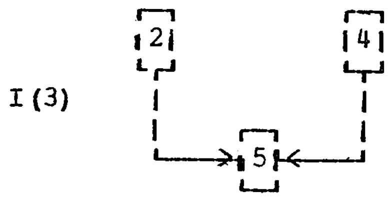


Figure 3.2 Directed Graph for Simple Program Node

possible source units (represented as QMSUs) for I(3) are {3,1;6,2}, {3,1;4,0}, {2,0;6,2}, and {2,0;4,0}; and, the possible sink units for I(3) are {7,3} and {5,3}. With four possible source unit combinations and two possible sink unit combinations, I(3) has eight possible combinations. If an exhaustive tree searching approach is used, then after seven regenerations, the following optimal tree is selected.



While a graph-theoretic approach provides an effective way to represent the program node, it provides no advantage in the determination of QMSUs which may participate as alternate source and sink units. Although the optimization of the above program node is given in two-dimensional space, the analysis of the general problem involves three-dimensional space. In the above example, alternate units are identified by an analysis of vertical operational arcs (GATE operations), and each micro-operation is identified by a tree level; however, the general problem requires the identification of equivalent QMSUs, and the identification of equivalent operations. For example, if the above program node is coded with an equivalent action at I(4),

```
I(1):    GATE     {2}      {3}
I(2):    GATE     {4}      {6}
I(3):    ADD     {3;6}     {7}
I(4):    ADD     {2;4}     {8}
I(5):    GATE     {7}      {5}
```

then the graphical representation would be as given in Figure 3.3.

Equivalent actions, I(3) and I(4), must be represented in the tree so that all alternate source and sink units may be identified. In order to represent equivalent actions, in two-dimensional space, it is necessary to perform the following functions:

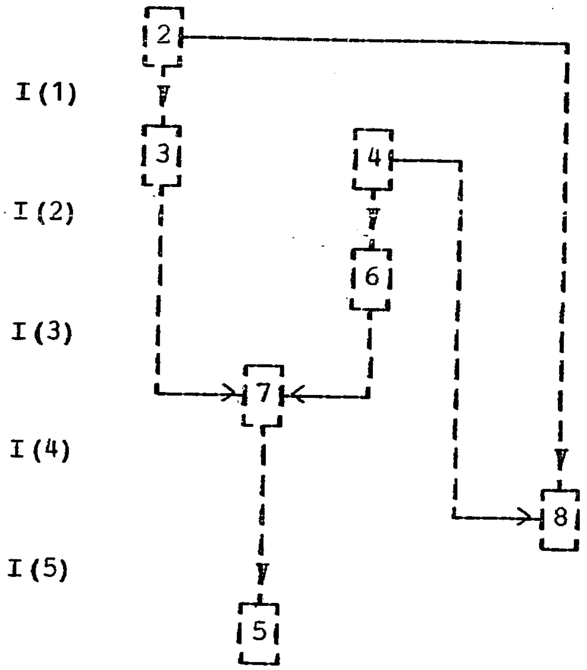


Figure 3.3 Directed Graph of Program Node With Equivalent Actions

1. With each action examined, the sub-graph which emanates from the source unit nodes is examined in order to identify equivalent predecessor actions.

2. A set of forward pointers is defined for the predecessor actions which are equivalent to the current action, and a set of backward pointers is defined for equivalent successor actions.

3. The current action's sink unit is identified as an alternate sink unit for the equivalent predecessor actions.

While the above functions are not impossible to perform, the graphical representation of all QMSUs which may participate as alternate source and sink units involves needless recursive analysis of predecessor and successor sub-graphs. Furthermore, there is no reduction of information when MO(i), i=1,2,...,N, are converted to their graphical representation, since Op(i) and c(i) must be maintained for the analysis of equivalent operations.

Although a graph-theoretic approach is feasible, the identification and representation of equivalent QMSUs and actions present needless computational complexity. In the next Chapter, it is shown that all alternate source and sink units can be identified, with a maximum of two examinations of each micro-operation in the program node. For an analysis of other approaches to similar combinatorics problems see [39].

3.5 Summary

The major objective of this Chapter has been to describe the framework for the study. The problems dealt with in the study were defined; the limitations and means of analysis were presented; and, the objectives of the study were discussed. In the next Chapter, a major problem of the study is undertaken: The identification of all possible forms of each micro-operation.

CHAPTER IV

IDENTIFICATION OF ALTERNATE ACTION FORMS

4.1 Introduction

The objective of this Chapter is to present and prove the correctness of an algorithm which identifies all possible forms of each action in a program node. As stated in Chapter 3, the variables of form are source and sink units. In Section 4.2, alternate source and sink units are defined as to type. A general description of the alternate unit identification algorithm is presented in Section 4.3 along with detailed descriptions of each pass in the algorithm. A proof of the correctness of each pass is offered in Section 4.4, and then a brief discussion of parallel actions is presented in Section 4.5.

4.2 Alternate Unit Types

It was found that there are three types of alternate source units: Those units which are not redefined (Current value) before the appearance of the action to which they are alternates (Type C); those units which are Redefined before the appearance of the action to which they are alternates (Type R); and, those units which are defined by the assignment of Alternate sink units (Type A).

The approach used to identify alternate sink units is to represent, for each action, those actions to which the sink unit definition may be assigned. From def 3.9, it can be seen that if sk(i) may be assigned to MO(j), then sk(j) may be assigned to MO(i); therefore, the representation of all actions to which sk(i) may be assigned is sufficient to identify all alternate sink units for MO(i), i=1,2,...,N. There are two types of alternate sink unit references: Those which represent Predecessor actions to which a sink unit may be assigned (Type P); and, those which represent Successor actions to which a sink unit may be assigned (Type S).

The conditions which define each unit type are given below; also, the information required to uniquely identify each unit is presented. It should be noted that the conditions described below define possible alternate units; the conditions of feasibility are defined in Chapter 5.

4.2.1 Alternate Source Unit Types

def. 4.1 Type C alternate source unit: (m,k) is a Type C alternate source unit for (t,s) at MO(i), $1 \le m,t \le NS$, $1 \le k,s,i \le N$,

iff $k < i$;

```
    MO(k) <=> MO(s);
    For j=k+1,k+2,...,i-1
        UA(j,m) ≤ 1.
```

Unique representation of (m,k) as a Type C alternate source unit is written as: (C,m,k).

<u>def</u>. 4.2 Type R alternate source unit: (m,k) is a Type R alternate source unit for (t,s) at MO(i), $1 \le m,t \le NS$, $1 \le k,s,i \le N$,

iff $k < i$;

MO(k) <=> MO(s);

For at least one j, $j=k+1,k+2,\ldots,i-1$

UA(j,m) > 1.

Unique representation of (m,k) as a Type R alternate source unit: (R,m,k).

<u>def</u>. 4.3 Type A alternate source unit: (m,k) is a Type A alternate source unit for (t,s) at MO(i), $1 \le m,t \le NS$, $1 \le k,s,i \le N$,

iff $k < i$;

sk(x) = m;

MO(x) <=> MO(k);

MO(k) <=> MO(s).

Unique representation of (m,k) as a Type A alternate source unit at MO(i) (assume MO(x) defines m): (A,m,k,x).

Alternate sink units may be identified by the representation of equivalent actions. The next section describes the two types of alternate sink unit references: Equivalent successor action references and equivalent predecessor action references.

4.2.2 Alternate Sink Unit Reference Types

def. 4.4 Type P alternate sink unit reference: MO(i) is a Type P alternate sink unit reference at MO(j), $1 \le i,j \le N$, iff $i < j$;

$i > 0$;

MO(i) <=> MO(j).

Unique representation of MO(i) as a Type P alternate sink unit reference at MO(j): (P,i).

def. 4.5 Type S alternate sink unit reference: MO(i) is a Type S alternate sink unit reference at MO(j), $1 \le i,j \le N$, iff $i > j$;

$j > 0$;

MO(i) <=> MO(j).

Unique representation of MO(i) as a Type S alternate sink unit reference at MO(j): (S,i).

Alternate source and sink units have been defined as to type. The next major section describes the organization and representation of alternate unit information.

4.3 Organization of Alternate Unit Information

The organization of the alternate unit information is in tabular form. The alternate unit assignment table (ALT) is used to represent prime and alternate source and sink units. The alternate unit identification algorithm, described in Section 4.4, constructs the ALT.

4.3.1 Alternate Unit Assignment Table (ALT)

The ALT is designed to represent all possible forms of each action by the specification of all alternate source and sink unit assignments. It is generally structured to contain the following: Unit references which may participate as alternate source units; and, action references to which sink units may be alternately assigned. A precise statement of the structure and referencing of the ALT is given next.

4.3.1.1 ALT Structure

With N actions, p source units per action (maximum), and one sink unit per action, the ALT is an N by (p+1) array. Each element in the array is divided into two parts: The prime unit specification and the alternate unit specification.

The prime source unit specification is an ordered pair, QMSU, which represents the source unit specified in the source form of the action. The prime sink unit specification is an ordered pair, (u,uid), where u is the identifier of the sink unit specified in the source form of the action, and uid is the index of the action. The prime unit fields for both the source and sink units are ordered pairs referenced in the ALT as scalars: ALT(i,psc(r)) references the rth prime source unit for MO(i), $1 \leq r \leq nsc(i,1)$; ALT(i,psk) references the prime sink unit for MO(i); and, ALT(i,psc) references all prime source units for MO(i).

The alternate source unit specification is an ordered triple if the unit is a Type C or Type R alternate, and an ordered quadruple if it is a Type A alternate. The order of values is as defined in Section 4.2.1. The alternate source unit field in the ALT is referenced as a vector of entries where each entry is an ordered set: ALT(i,asc(r,j,k)) references the kth value in the jth ordered set which is an alternate for prime source unit ALT(i,psc(r)), $1 \le r \le nsc(i,1)$, $1 \le j \le nALT(i,asc(r))$, $1 \le k \le 4$. ALT(i,asc(r,j)) references the jth ordered set; ALT(i,asc(r)) references all ordered sets which are alternate for ALT(i,psc(r)); and, ALT(i,asc) references all ordered sets which are alternate source units in MO(i).

The alternate sink unit specification is an ordered pair whose values are as defined in Section 4.2.2. The alternate sink unit field is referenced as a vector of ordered pairs: ALT(i,ask(j,k)) references the kth value in the jth ordered pair which represents an alternate sink unit assignment for MO(i), $1 \le j \le nALT(i,ask(j))$, $1 \le k \le 2$. ALT(i,ask(j)) references the jth ordered pair for MO(i); and, ALT(i,ask) references all ordered pairs which represent alternate sink unit assignments for MO(i). To clarify the structure and referencing of fields within the ALT, an example is given.

4.3.1.2 ALT Example

As an example of the ALT structure, Table 4.1 gives the ALT generated for the program node in Figure 3.1. To review the notation and aid in understanding the ALT, the I(3) entry in Table 4.1 is discussed. The first prime source unit for MO(3) is {3}, the second is {6}. Relative to MO(3), units {3} and {6} are last defined at MO(1) and MO(2) respectively. Thus, the ordered pairs, (3,1) and (6,2), are QMSU's which represent prime source units for MO(3). In terms of referencing these pairs as ALT elements, ALT(3,psc(1)) = (3,1) and ALT(3,psc(2)) = (6,2). Similarly, the prime sink unit entry, (7,3), references unit {7} defined at MO(3); and, ALT(3,psk) = (7,3).

The first alternate source unit entry, (C,2,0), means that unit {2} defined at MO(0) [on entry] is a Type C alternate for unit {3} at MO(3) [recall, Type C means unit {2} is not redefined between MO(0) and MO(3)]. In terms of referencing the ALT elements, ALT(3,asc) = (C,2,0) (C,4,0); ALT(3,asc(1)) = (C,2,0), and ALT(3,asc(2)) = (C,4,0); ALT(3,asc(2,1)) = (C,4,0); and, ALT(3,asc(2,1,2)) = 2.

The alternate sink unit entry, (S,6), means that MO(6) is a Type S alternate sink unit reference for MO(3); and, ALT(3,ask(1)) = (S,6). That is, MO(3) and MO(6) are equivalent and the unit defined at MO(3) may (possibly) be defined at MO(6), and vice versa.

To review notation developed in Chapter 3, a symbolic qualification of MO(3) is presented.

MO(3) <=> MO(6) [MO(3) and MO(6) are equivalent].

Op(3) = ADD [operation for MO(3)].

sk(3) = {7} [sink unit for MO(3)].

c(3) = ∅ [no control bits for MO(3)].

sc(3) = {3;6} {3;4} {2;6} {2;4} [all possible source unit combinations, expressed as QMSUs, for MO(3)].

nsc(3) = 4 [number of possible source unit combinations].

sc(3,1) = {3;6} [first (source-level) source unit combination for MO(3)].

nsc(3,1) = 2 [number of source units in first source unit combinations for MO(3)].

sc(3,4,2) = {4} [second unit in fourth source unit combination for MO(3)].

(6,2) <=> (4,0) [unit {6} defined at MO(2) is equivalent to unit {4} defined at MO(0)].

Table 4.1 ALT for Program Node Given in Figure 3.1

	SOURCE 1		SOURCE 2		SINK	
Index	prime	alternate	prime	alternate	prime	alternate
I(1)	(2,0)	Ø	Ø	Ø	(3,1)	Ø
I(2)	(4,0)	Ø	Ø	Ø	(6,2)	Ø
I(3)	(3,1)	(C,2,0)	(6,2)	(C,4,0)	(7,3)	(S,6)
I(4)	(1,0)	Ø	(2,0)	(C,3,1)	(2,4)	Ø
I(5)	(2,4)	Ø	(4,0)	(C,6,2)	(8,5)	(S,7)
I(6)	(7,3)	(A,5,3,6)	Ø	Ø	(5,6)	(P,3)
I(7)	(2,4)	Ø	(6,2)	(C,4,0)	(9,7)	(P,5)
I(8)	(7,3)	(A,7,6,3) (A,5,3,6) (C,5,6)	(3,1)	(R,2,0)	(3,8)	Ø

The next major section describes the algorithm whose primary purpose is the construction of the ALT.

4.4 Alternate Unit Identification Algorithm: Description

The ALT is constructed by the algorithm described below. It is divided into two passes. Pass 1 is responsible for the identification of Type C and Type R alternate source units and Type P and Type S alternate sink units. Pass 2 identifies all Type A alternate source units. Pass 1 requires only one examination of each action in the program node; and, Pass 2 requires only one examination of each alternate sink unit entry defined in Pass 1. Actions are examined in sequence within the node and within the ALT.

In order to represent all possible combinations of Type C and Type R alternate source units and Type P and Type S alternate sink units, with only one examination of each action, the algorithm must provide the following information upon examination of MO(i), $1 \leq i \leq N$: sk(i) must be "matched" with all QMSUs previously defined to contain the same value; and, all possible source unit combinations of MO(i) which produce results identical to previous actions must be identified. Concisely, equivalent sink units are combined to identify alternate source units; equivalent actions are combined to identify alternate sink units. The next section describes the algorithm's general approach to the identification of equivalent units and equivalent actions.

4.4.1 General Description of Algorithm Approach

The algorithm proposes the unique identification of actions by the application of a hashing function [28] to (Op(i),sc(i),c(i)). The hashing function may, for example, be a Boolean operator function or any residue modulo(n) arithmetic technique; also, the hash table may have only a prime area or it may be organized as a linked list. The hashing function selected is of little importance to this study, as it would likely be dependent upon average node size, encoding structure, hashing source image, and operation distribution factors. For the purposes of this study, it is assumed that each unique hash image, (Op(i),sc(i,j),c(i)), when hashed, generates a unique hash table address.

4.4.1.1 Operation Hash Table (OHT)

The algorithm constructs an operation hash table, OHT, and a unit hash address table, UHA. Each element in the OHT is a vector of ordered pairs, QMSUs, which are equivalently defined by the action that corresponds to the element address; exception: OHT(NOP,(i,j),0) contains a list of OHT addresses where unit {i}, as defined by MO(j), is represented. For example, the following code is presented (note that I(2) and I(3) are equivalent actions).

I(1):	GATE	{3}	{8}
I(2):	ADD	{8;2}	{7}
I(3):	ADD	{3;2}	{6}

If the respective OHT addresses are as follows,

(GATE,(3,0),0) = 31

(ADD,(8,1;2,0),0) = 16

(ADD,(3,0;2,0),0) = 28

then (8,1) ∈ (31)

(7,2),(6,3) ∈ (16)

(7,2),(6,3) ∈ (28)

and OHT(NOP,(8,1),0) = 31

OHT(NOP,(7,2),0) = 16 28

OHT(NOP,(6,3),0) = 16 28

The function of the OHT is to represent those QMSU's which are equivalently defined. In order to retrieve equivalent unit information from the OHT, a method is required for accessing the OHT by unit reference. The unit hash address table (UHA), described in the next section, provides a way in which a set of equivalent units may be found by accessing only one member in the set.

4.4.1.2 Unit Hash Address Table (UHA)

With NS memory sub-units, the unit hash address table (UHA) is a 2 by NS array where UHA(1,i) contains the index of the action which last defined unit {i}; UHA(2,i) contains the OHT address which corresponds to the source form of the action which last defined unit {i}. For example, if unit {6} is last defined at MO(9) by (ADD,(3,0;4,2),0), and if (ADD,(3,0;4,2),0) hashes to OHT address 16, then UHA(1,6) = 9 and UHA(2,6) = 16.

The memory management techniques for the storage and retrieval of the OHT and UHA elements are not presented. Such techniques are dependent upon specific machine and main-memory characteristics and would, therefore, vary with different configurations.

A final note on the hashing function: Ordering of the source units, where primitive operations conform to commutative laws, eliminates the chance of nonmatches on action forms which have the same result. For example, source units for primitive operations such as ADD should have a defined order of appearance in the hashing source image.

A detailed description of each algorithm pass is presented next. For the pass descriptions and proofs, the following qualifications are given: OHT'i' represents the OHT address generated by hashing (Op(i),sc(i,j),c(i)); OHT(i) represents the vector of ordered pairs at OHT'i'; OHT[i] represents the vector of ordered pairs where i is a pre-defined OHT address; the symbol, ▪ , is used to represent concatenation; and, (x ≠ y) represents all x except y where x ∩ y ≠ ∅ is possible.

4.4.2 Pass 1: Detailed Description

The major portion of the algorithm is dedicated to Pass 1. As stated earlier, Pass 1 identifies all Type C and Type R alternate source units and all Type P and Type S alternate sink units for each action. Pass 1 is divided into three phases.

Phase 1.1: Identifies all Type C and Type R alternate source units for each action.

Phase 1.2: Determines alternate sink unit references for unmodified data-flow actions; e.g., a GATE micro-operation where the control field is null.

Phase 1.3: Determines alternate sink unit references for actions which are not unmodified data-flow actions.

4.4.2.1 Pass 1, Phase 1.1 (Figure 4.1)

Phase 1.1 initializes the UHA and OHT before the program node is examined; identifies prime and alternate source units (Types C and R) for each action; determines the hash address of the current action with its prime source units; and, updates the UHA for the current action.

Step 1. The OHT and UHA are initialized before the program node is examined. Each unit is identified as if it were defined by a trivial SHIFT action; i.e., Op(0) = SHIFT, sc(0,1) = j, sk(0) = j, c(0) = Ø. UHA(1,j) = 0 for all j so that references to units defined in a predecessor node are distinct from units defined in the current node.

UHA(2,j) = OHT'SHIFT,(j,0),0';

OHT(SHIFT,(j,0),0) = (j,0); and,

OHT(NOP,(j,0),0) = OHT'SHIFT,(j,0),0'. It is noted that if the analysis is performed on a regional basis, the UHA(2,j) is left unchanged for those units which are busy on entry into a node.

Step 2. Prime and alternate source units are identified and placed in the ALT; and, each prime source unit is augmented with its current index of definition.

For example, if the current action is

I(9): ADD {3;4} {6}

and the related UHA and OHT values are as follows,

			Units		
	1	2	3	4	5
UHA	2	0	4	5	3
	16	10	12	19	21

OHT(12) = (2,0) (1,1) (3,4)

OHT(19) = (5,3) (4,5)

then, after Step 2 is completed,

ALT(9,psc(1)) = (3,4)   ALT(9,asc(1)) = (C,2,0) (R,1,1)

ALT(9,psc(2)) = (4,5)   ALT(9,asc(2)) = (C,5,3)

and, the field X contains the operation and QMSUs given as prime source units:

X = ADD,(3,4;4,5)

The field X is used in Step 3 to form the hash source image.

Step 3. The hash address of the current action is determined; the prime sink unit is placed in the ALT; and, the UHA entry for the prime sink unit is updated to reflect its new index of definition. For the example given in Step 2, the following values are defined in Step 3:

B = OHT'ADD,(3,4;4,5),0'

UHA(1,6) = 9

UHA(2,6) = B

ALT(9,psk) = (6,9)

The primary function of Phase 1.1 is the retrieval, from the OHT, of Type C and Type R alternate source unit information. The variables of form still to be determined are Type A alternate source units, and Type P and Type S alternate sink unit references. When a unit is defined by an unmodified data-flow action, alternate sink unit references are easily identified by an examination of QMSUs which are equivalent to the prime source unit given in the action. The next section describes Phase 1.2, whose major function is the identification of alternate sink unit references for an unmodified data-flow action.

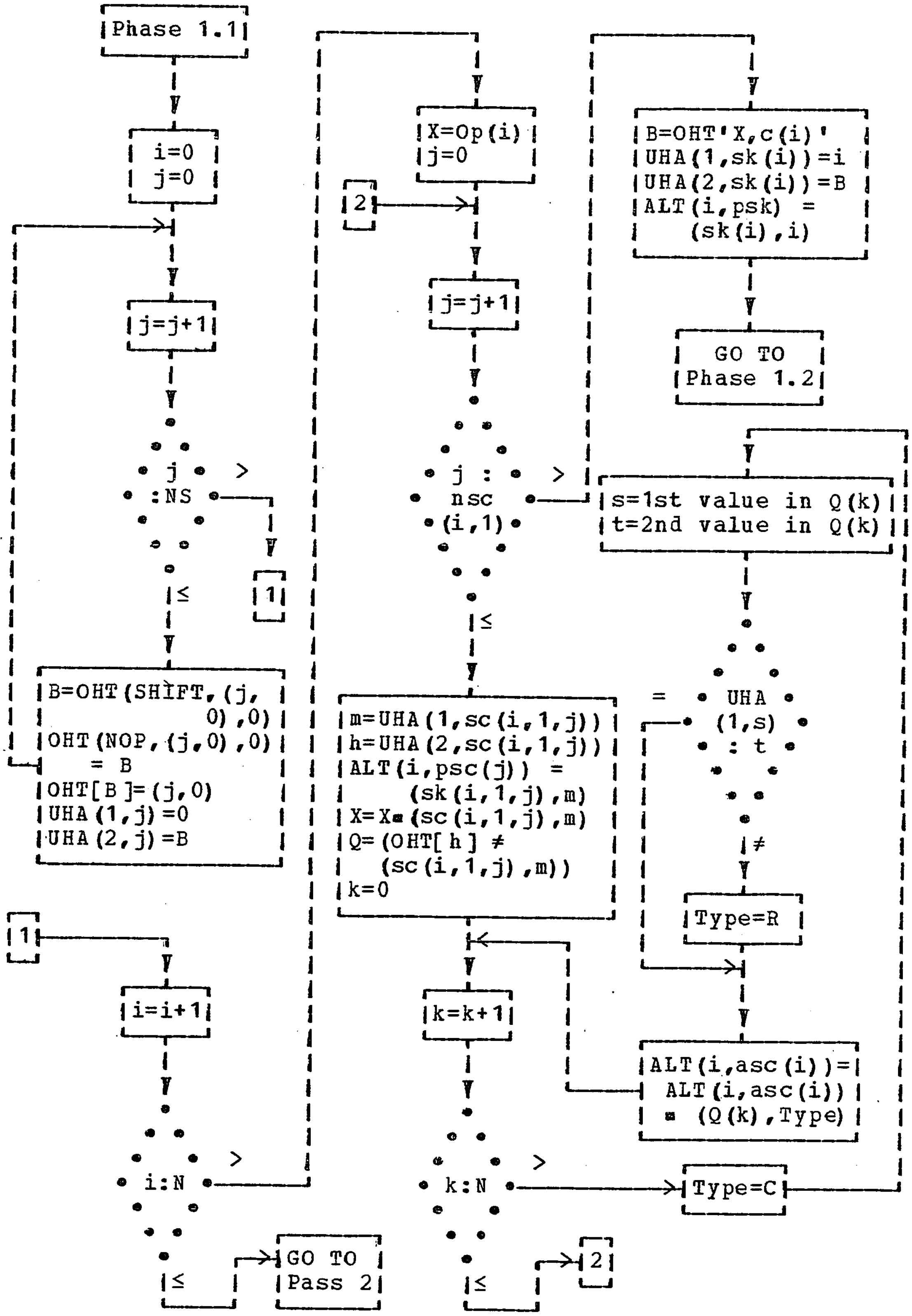


Figure 4.1 Flow Chart for Pass 1, Phase 1.1: Type C and Type R Alternate Source Unit Identification

4.4.2.2 Pass 1, Phase 1.2 (Figure 4.2)

Phase 1.2 determines if the current action, MO(i), is an unmodified data-flow action; if so, sk(i) is made equivalent to the prime source unit and all QMSUs equivalent to the prime source unit. Then, sk(i) is assigned as an alternate sink unit reference.

Step 1. If the action is not an unmodified data-flow action, then Phase 1.2 is bypassed and Phase 1.3 is called; i.e., if sk(i) <=> sc(i,1) by an unmodified data-flow action, Phase 1.2 is executed, otherwise Phase 1.3 is executed.

Step 2. Since an unmodified data-flow action defines the sink unit to be equivalent to the source unit, the sink unit must be made equivalent to the source unit and all previously defined QMSUs which are equivalent to the source unit. This is accomplished when the OHT address, UHA(2,sc(i,1)), which corresponds to the source form of the action that defined (sc(i,1)) is placed in UHA(2,sk(i)); and then, (sk(i),i) is placed at all OHT addresses which contain sc(i,1). OHT(NOP,(sc(i,1),UHA(1,sc(i,1))),0) contains the list of OHT addresses where sc(i,1) is contained. This step also constructs a vector, E, which contains all QMSUs, sc(i,1) is included, that are equivalent to sc(i,1). This list is used in Step 3.

As an example of the operations performed, if I(5) is the current action in the following program node (note that I(2), I(4), and I(5) are equivalent actions),

```
    I(1):    SUB     {4;7}    {2}
    I(2):    ADD     {1;2}    {5}
    I(3):    SUB     {4;7}    {8}
    I(4):    ADD     {1;8}    {3}
    I(5):    GATE     {5}     {1}
```

```
and if OHT'ADD,(1,0;2,1),0' = 23
       OHT'ADD,(1,0;8,3),0' = 36
then   OHT(23) = (5,2) (3,4)
       OHT(36) = (5,2) (3,4)
       OHT(NOP,(5,2),0) = 23 36
```

Step 2 d fines the following values, while I(5) is processed:

```
           D = 23 36
           E = (3,4) (5,2)
     OHT(23) = (5,2) (3,4) (1,5)
     OHT(36) = (5,2) (3,4) (1,5)
    UHA(2,1) = 23
```

Step 3. This step uses the vector E formed in Step 2 to identify those actions to which sk(i) is an alternate sink unit. Since sk(i) of an unmodified data-flow action is an alternate sink unit for sc(i,1) and all QMSUs equivalent to sc(i,1), sk(i) is an alternate sink unit for all actions referenced by QMSUs in E. After I(5), in the example given in Step 2, is operated upon in Step 3, the following alternate sink unit references are defined:

ALT(2,ask) = (S,4) (S,5)

ALT(4,ask) = (P,2) (S,5)

ALT(5,ask) = (P,2) (P,4)

The reference addresses for (sk(i),i) are also defined in Step 3. For the example in Step 2,

OHT(NOP,(1,5),0) = 23 36

If the current action is not an unmodified data-flow action, then alternate sink unit references are determined by another method. This method identifies all units defined by actions equivalent to the action currently under examination. The general technique used in this method is to hash all possible source unit combinations for the current action; thus, all units defined equivalently to the current action's sink unit are identified. Phase 1.3 is responsible for the implementation of this technique.

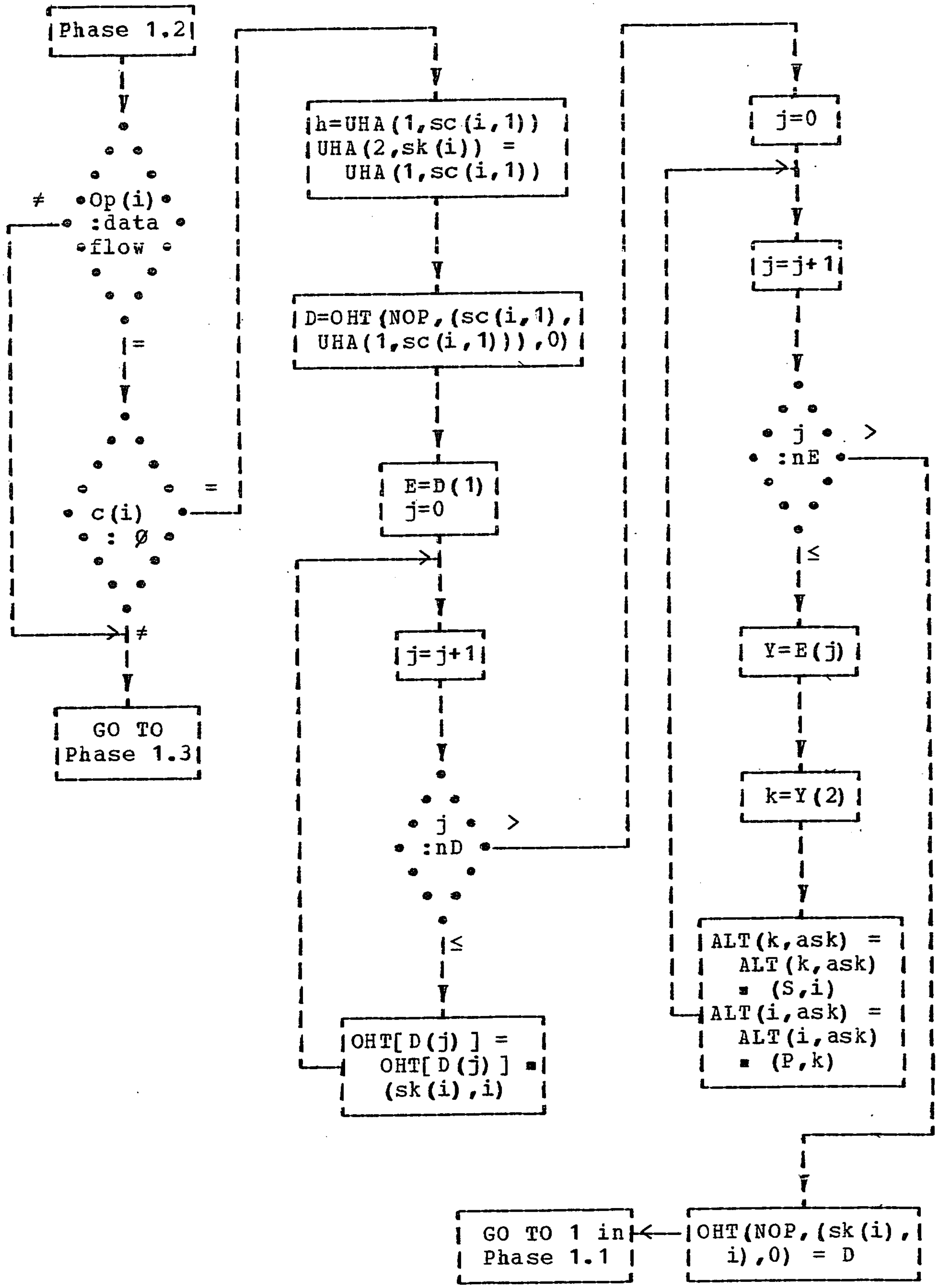

Phase 1.2
≠
Op(i) :data flow
=
c(i) : ∅
=
≠
GO TO Phase 1.3
h=UHA(1,sc(i,1))
UHA(2,sk(i)) = UHA(1,sc(i,1))
D=OHT(NOP,(sc(i,1), UHA(1,sc(i,1))),0)
E=D(1)
j=0
j=j+1
j :nD
>
≤
OHT[D(j)] = OHT[D(j)] ∎ (sk(i),i)
j=0
j=j+1
j :nE
>
≤
Y=E(j)
k=Y(2)
ALT(k,ask) = ALT(k,ask) ∎ (S,i)
ALT(i,ask) = ALT(i,ask) ∎ (P,k)
OHT(NOP,(sk(i), i),0) = D
GO TO 1 in Phase 1.1

Figure 4.2 Flow Chart for Pass 1, Phase 1.2: Unmodified Data-Flow action

4.4.2.3 Pass 1, Phase 1.3 (Figure 4.3)

Phase 1.3 determines the hash address of each possible source unit combination in MO(i); identifies all previously defined QMSUs which are equivalent to sk(i); MO(i) is assigned as an alternate sink unit reference at all equivalent predecessor actions; and, the OHT is updated to reflect the changes created by MO(i).

Step 1. The hash address of each possible source unit combination is determined. This step constructs the vector D which contains a list of all hash addresses generated from (Op(i),sc(i),c(i)); and, the vector E which contains a list of all unique QMSUs stored at the addresses specified in D (i.e., a list of equivalent QMSUs).

After step 1 has operated upon I(6) in the following program node (note that I(2), I(4), and I(6) are equivalent actions),

I(1):	SUB	{2;1}	{3}
I(2):	ADD	{3;4}	{6}
I(3):	GATE	{3}	{5}
I(4):	ADD	{4;5}	{8}
I(5):	GATE	{4}	{7}
I(6):	ADD	{5;7}	{3}

if OHT'ADD,(3,1;4,0),0' = 13

OHT'ADD,(4,0;5,3),0' = 26

OHT'ADD,(5,3;7,5),0' = 52

then D = 52 26 13

E = (3,6) (6,2) (8,4)

Step 2. The reference addresses in the OHT, OHT(NOP,E(j),0), are updated for all equivalent units identified in Step 1; also, previous actions to which (sk(i),i) is an alternate sink unit are identified. Since E contains equivalent QMSUs defined at addresses specified in D, reference addresses are updated by the assignment of D to OHT(NOP,E(j),0) for each E(j), j=1,2,...,nE. E(1) = (sk(i),i), and E(2),E(3),...,E(nE) contains a list of previously defined QMSUs which are equivalent to (sk(i),i); therefore, each action which defined E(j) is a Type P alternate sink unit reference for MO(i), and MO(i) is a Type S alternate sink unit reference for each action which defined E(j), j=2,3,...,nE.

For the example given in Step 1, Step 2 generates the following values:

OHT(NOP,(3,6),0) = 13 26 52

OHT(NOP,(8,4),0) = 13 26 52

OHT(NOP,(6,2),0) = 13 26 52

ALT(2,ask) = (S,4) (S,6)

ALT(4,ask) = (P,2) (S,6)

ALT(6,ask) = (P,2) (P,4)

Step 3. This step, the final one in Pass 1, updates the OHT to contain QMSUs equivalently defined by the current action. Since D contains a list of all OHT addresses generated by the current action, and E contains all QMSUs equivalently defined by the current action, the OHT is updated by the placement of E in each OHT location specified by D.

For the example given in Step 1, the following OHT values are defined in Step 3:

OHT(13) = (3,6) (8,4) (6,2)

OHT(26) = (3,6) (8,4) (6,2)

OHT(52) = (3,6) (8,4) (6,2)

Phase 1.2 and Phase 1.3 insure that all Type P and Type S alternate sink unit references are identified for each action. With alternate sink unit references determined, it is possible to identify Type A alternate source units. The next section describes Pass 2, which uses the alternate sink unit references to determine Type A alternate source units.

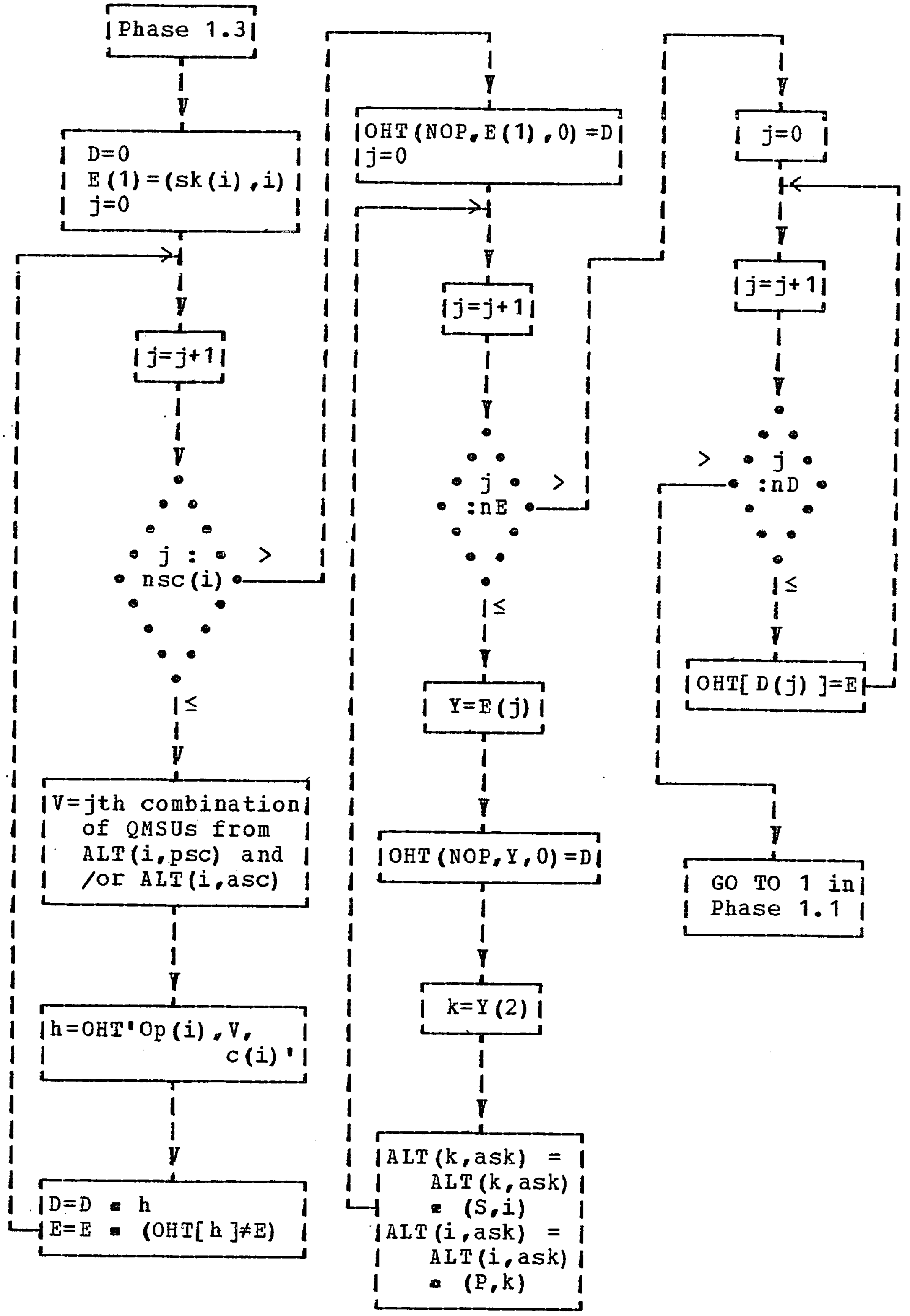


Figure 4.3 Flow Chart for Pass 1, Phase 1.3: Alternate Sink Unit Identification

4.4.3 Pass 2: Detailed Description

Pass 2 is designed to identify all Type A alternate source units by an examination of Type P and Type S alternate sink units. Phase 2.1 examines the alternate sink unit specification in the ALT, in order to identify Type A alternate source units.

4.4.3.1 Pass 2, Phase 2.1 (Figure 4.4)

Phase 2.1, the final phase in the algorithm, examines each action represented in the ALT, from first to last, to determine if alternate sink unit references are present. If so, these references are used in the determination of Type A alternate source units.

Step 1. Selects the alternate sink unit entries for each action (sender), and then determines the prime sink unit for the action (receiver) to which an alternate sink unit may be assigned.

Step 2. Examines the ALT, in sequence, from the receiver to the last action represented in the ALT. If the prime sink unit for the receiver action is a prime or alternate (Type C or Type R) source unit, then the prime sink unit of the sender is identified as a Type A alternate sink unit.

Each pass of the algorithm has been described in detail. The next major section offers proof that the algorithm identifies all alternate and equivalent action forms.

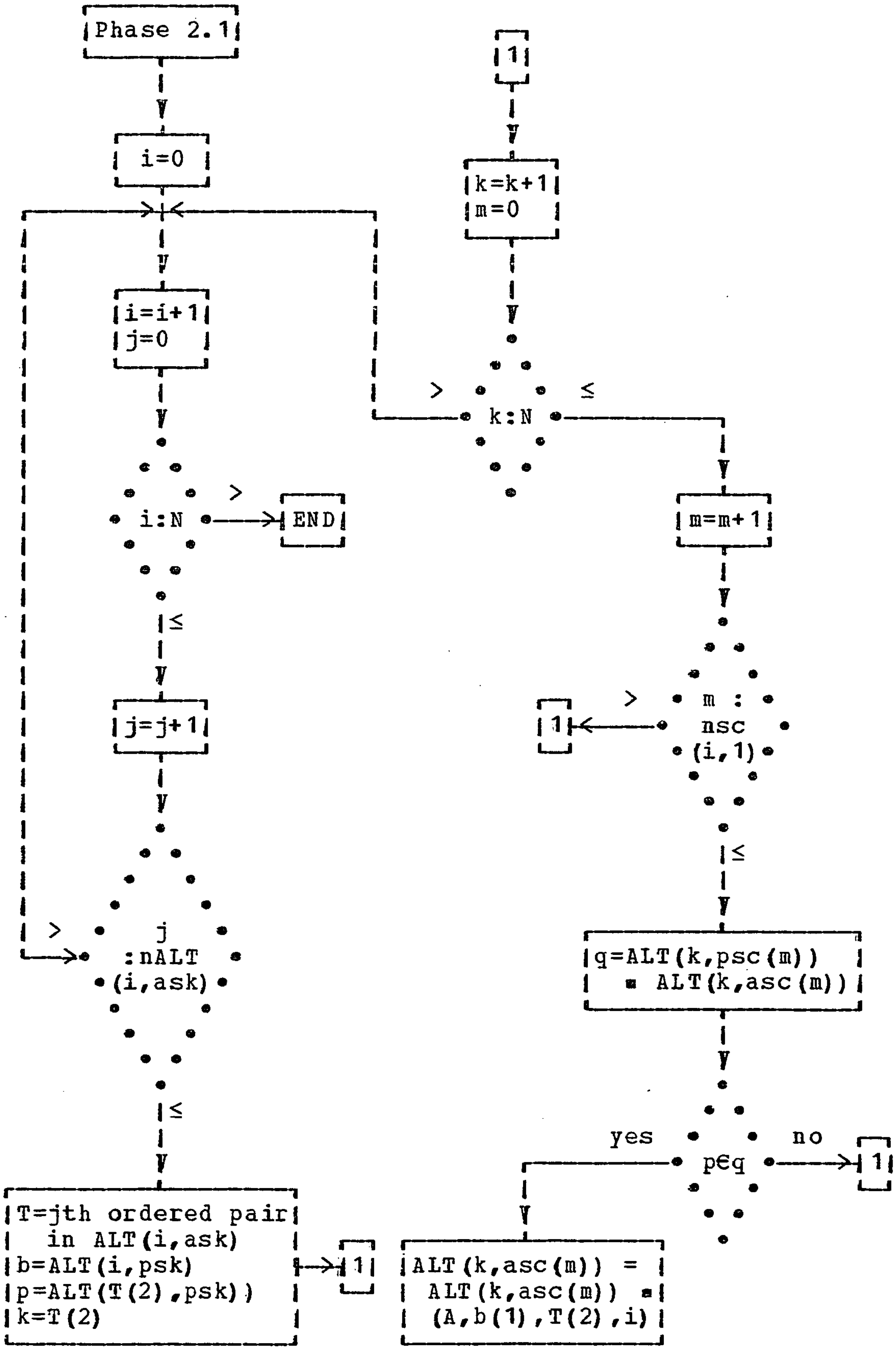


Figure 4.4 Flow Chart for Pass 2, Phase 2.1: Type A Alternate Source Unit Identification

4.5 Alternate Unit Identification Algorithm: Proof

The general approach used to prove that the algorithm identifies all alternate units is to state and prove assertions which support the intention of each pass. For Pass 1, proof is given that all Type C and Type R alternate source units and all Type P and Type S alternate sink units are identified; for Pass 2, proof is given that all Type A alternate source units are identified. Before a proof is offered for each pass, three theorems are presented which are basic to the algorithm. Throughout the proofs, the acronym "UDF" is generally used in place of the more verbose "unmodified data-flow action".

Theorem 4.1 Given N actions in a program node: If MO(i) <=> MO(j) and i < j, then (sk(i),i) is defined at MO(j) by (Op(j),sc(j),c(j)), $1 \le i < N$, $1 < j \le N$.

Proof:

From def 3.7, for MO(i) and MO(j) ≠ UDF,

OP(i) = OP(j), c(i) = c(j), and sc(i,1) <=> sc(j,1) [recall, sc(i,1) references the 1st source unit combination at MO(i)];

and, since equivalent units are transitive, sc(i) <=> sc(j).

Since i < j and equivalent units are qualified by their index of definition, sc(i) ∈ sc(j).

If $sc(i) \in sc(j)$ and $MO(i) <=> MO(j)$, then $sc(i) \notin sc(j)$ defines an additional set of source unit combinations which identify action forms that may define $(sk(i),i)$.

For either $MO(i)$ or $MO(j) = UDF$, $(sk(i),i)$ is obviously defined at $MO(j)$; therefore, where $MO(i) <=> MO(j)$ and $i < j$, $(sk(i),i)$ is defined at $MO(j)$ by $(Op(j),sc(j),c(j))$, $1 \leq i < N$, $1 < j \leq N$.

Q.E.D.

Theorem 4.2 Given N actions in a program node: If all Type P alternate sink units are identified at $MO(j)$, then all Type S alternate sink units may be identified at $MO(i)$, $i=2,3,\ldots,N$, $j=1,2,\ldots,N-1$.

Proof:

From *def* 4.4, $MO(j)$ is a Type P alternate sink unit reference at $MO(i)$

iff $i < j$ and

$MO(i) <=> MO(j)$;

from *def* 4.5, $MO(j)$ is a Type S alternate sink unit reference at $MO(i)$

iff $i < j$ and

$MO(i) <=> MO(j)$.

Q.E.D.

Theorem 4.3 The identification of Type C and Type R alternate source units is sufficient to identify all Type P and Type S alternate sink unit references in a program node.

Proof:

From <u>def</u> 4.1 and 4.2, it can be seen that all Type C and Type R alternate source units are identified only if all equivalent predecessor actions are identified. From Theorem 4.2, the identification of all equivalent predecessor actions is sufficient to identify all equivalent actions; therefore, the identification of all Type C and Type R alternate source units is sufficient to identify all Type P and Type S alternate sink unit references in a program node.

Q.E.D.

<u>Corollary</u> 4.3 The identification of all Type C and Type R alternate source units is sufficient to identify all equivalent actions in a program node.

Proof:

Clearly, Type A alternate source units cannot form new action equivalencies; therefore, Corollary 4.3 is immediate from Theorems 4.2 and 4.3.

Q.E.D.

The theorems presented are used throughout the proofs given in the next two sections. The following section offers proof that Pass 1 of the algorithm identifies all Type C and Type R alternate source units and all Type P and Type S alternate sink units.

4.5.1 Pass 1: Proof

To show that Pass 1 identifies all Type C and Type R alternate source units and all Type P and Type S alternate sink units, the following assertions are made (assume N actions in a program node and NS memory sub-units).

Assertion 4.1 Each possible and different form for MO(i) is uniquely identified, i=1,2,...,N.

Proof:

The hashing function is applied to each (Op(i),sc(i,j),c(i)) for j=1,2,...,nsc(i) [nsc(i) means number of source unit combinations for MO(i)]; and, where commutative laws apply to Op(i), there is a defined order of representation for sc(i,j). Therefore, since each unique (Op(i),sc(i,j),c(i)) hashes to a unique address, each possible and different form of each MO(i) is uniquely identified by its corresponding OHT address.

Q.E.D.

Assertion 4.2 If MO(i) redefines unit {k}, all references to previous definitions of unit {k} are differentiated from all references to the MO(i) definition of unit {k}, 1≤i≤N, 1≤k≤NS.

Proof:

Since each unit is augmented by its index of definition, references to the MO(i) definition of unit {k} are distinct from references to previous definitions of unit {k}; i.e., the MO(i) definition of unit {k} is referenced as (k,i).

Q.E.D.

<u>Assertion</u> 4.3 UHA(1,k) contains the most recent index of definition for unit {k}, $1 \le k \le NS$.

Proof:

Actions are examined in first to last sequence within the program node and all units have an index of definition = 0 upon entry into the program node.

```
For i=1,2,...,N
    UHA(1,sk(i)) := i;
```

therefore, UHA(1,k) always contains the most recent (current) index of definition for unit {k}, $1 \le k \le NS$.

Q.E.D.

<u>Assertion</u> 4.4 For the examination of MO(t) where MO(t) = unmodified data-flow action, the generation of entries at OHT'Op(t),sc(t),0' is unnecessary, t=1,2,...,N.

Proof: (assume sc(t,1) = {e})

[recall, OHT(i) means all elements at OHT address generated by hashing i; OHT'i' means OHT address generated by hashing i; OHT[i] means elements at OHT address i; ■ means concatenation; and, (x ≠ y) means all x except y]

When MO(t) = UDF and sc(t,1) is represented as (e,UHA(1,e)):

```
    D := OHT(NOP,(e,UHA(1,e)),0)
    where OHT(NOP,(e,UHA(1,e)),0) contains a complete list
    of OHT addresses that correspond to action forms which
    may define (e,UHA(1,e)).
    For m=1,2,...,nD
        OHT[D(m)] := OHT[D(m)] ■ (sk(t),t);
    thus, each OHT entry which corresponds to an action form
    that may define (e,UHA(1,e)) is updated to reflect the
    equivalent definition of (sk(t),t).
    OHT(NOP,(sk(t),t),0) := D;
    thus, OHT(NOP,(sk(t),t),0) contains a complete list of
    OHT addresses which correspond to action forms that may
    define (e,UHA(1,e)) and (sk(t),t).
    UHA(2,sk(t)) := UHA(2,e);
    thus, OHT[UHA(2,sk(t))] contains a complete list of
    QMSUs which are equivalent to (e,UHA(1,e)) and
    (sk(t),t).
```

If OHT(NOP,(e,UHA(1,e)),0) contains a complete list of OHT addresses that correspond to action forms which may define (e,UHA(1,e)); and, if OHT[UHA(2,e)] contains a complete list of QMSUs which are equivalent to (e,UHA(1,e)), then (sk(t),t) is made equivalent to (e,UHA(1,e)) and all QMSUs equivalent to (e,UHA(1,e)). Therefore, for the examination of MO(t) where MO(t) = UDF, the generation of entries at OHT'OP(t),sc(t),0' is not required for the identification of equivalencies, t=1,2,...,N.

Q.E.D.

<u>Assertion</u> 4.5 For the examination of MO(t) where MO(t) ≠ unmodified data-flow action, OHT(NOP,(i,j),0) contains a complete list of OHT addresses that correspond to action forms which may define (i,j), 1≤i≤NS, 1≤j≤N, t=j,j+1,...,N.

Proof:

When (i,j) is defined at MO(j) and MO(j) ≠ UDF:

OHT(NOP,(i,j),0) := D

where D := D ▪ OHT'Op(j),sc(j,m),c(j)'

for m=1,2,...,nsc(j).

Thus, for the examination of MO(t) where t = j, OHT(NOP,(i,j),0) is complete, 1≤i≤NS, 1≤j≤N.

When (k,s) is defined at MO(s), s > j, MO(s) <=> MO(j), and MO(s) ≠ UDF, 1≤k≤NS:

OHT(NOP,(i,j),0) := D

where D := D ▪ OHT'Op(s),sc(s,m),c(s)'

for m=1,2,...,nsc(s).

From Theorem 4.1, sc(j) ∈ sc(s), and sc(j) Ø sc(s) defines an additional set of source unit combinations which identify action forms that may define (i,j);

therefore, OHT(NOP,(i,j),0) ⊂ D. (D ≠ OHT(NOP,(i,j),0)) represents all OHT addresses which correspond to action forms identified by (Op(s),sc(j) ∅ sc(s),c(s)). From Assertion 4.3, no OHT addresses are generated when MO(i) = UDF; therefore, for an examination of MO(t) where t=j+1,j+2,...,N, OHT(NOP,(i,j),0) is complete, 1≤i≤N, t=j+1,j+2,...,N.

Clearly, for the examination of MO(t) for t = j and t=j+1,j+2,...,N, OHT(NOP,(i,j),0) is complete; therefore, for the examination of MO(t) where MO(t) ≠ UDF, OHT(NOP,(i,j),0) contains a complete list of OHT addresses that correspond to action forms which may define (i,j), 1≤i≤NS, 1≤j≤N, t=j,j+1,...,N.

Q.E.D.

<u>Assertion</u> 4.6 For the examination of MO(t) where (i,j) is defined at MO(j) and MO(j) = unmodified data-flow action, OHT(NOP,(i,j),0) contains a complete list of OHT addresses which correspond to action forms that may define (i,j), 1≤i≤NS, 1≤j≤N, t=j,j+1,...,N.

Proof:

When MO(s) <=> MO(j), s > j, MO(j) = UDF, and MO(s) ≠ UDF:

```
OHT(NOP,E(j),0) := D
where D := D ▪ OHT'Op(s),sc(s,m),c(s)'
    for m=1,2,...,nsc(s);
```

and, E := E ▪ (OHT[D(m)] ≠ E)

for m=1,2,...,nD;

for j=1,2,...,nE.

From Assertion 4.3, (sk(j),j) ∈ E; therefore, for MO(s) <=> MO(j), s > j, MO(j) = UDF, and MO(s) ≠ UDF, OHT(NOP,(i,j),0) is complete.

When (i,j) is defined at MO(j) and MO(j) = UDF:

OHT(NOP,(i,j),0) := OHT(NOP,(sc(j,1),UHA(1,sc(j,1))),0). Thus, for the examination of MO(j) where MO(j) = UDF, the completeness of OHT(NOP,(i,j),0) is dependent upon the completeness of OHT(NOP,(sc(j,1),UHA(1,sc(j,1))),0). It can be seen that convergence will occur (i.e., completeness can be proven) if the extreme dependency case is considered (i.e., a program node which contains only unmodified data-flow actions). For the extreme case, there exists a (z,0) which is defined at MO(0) and, by the initialization in Phase 1.1, MO(0) ≠ UDF; therefore, OHT(NOP,(z,0),0) can be proven to contain a complete list of OHT addresses that correspond to action forms which may define (z,0).

For the examination of MO(t) where (i,j) is defined at MO(j) and MO(j) = UDF, OHT(NOP,(i,j),0) contains a complete list of OHT addresses which correspond to action forms that may define (i,j), t = j and t=j+1,j+2,...,N, 1≤i≤NS, 0≤j≤N.

Q.E.D.

Corollary 4.6A(1) From Assertions 4.5 and 4.6:
For the examination of MO(t) where (i,j) is defined at MO(j), OHT(NOP,(i,j),0) contains a complete list of OHT addresses that correspond to action forms which may define (i,j), $1\le i\le NS$, $1\le j\le N$, $t = j,j+1,\ldots,N$.

Corollary 4.6A(2) From Assertions 4.5 and 4.6:
For the examination of MO(t) where (i,j) is defined at MO(j), OHT(NOP,(i,j),0) contains no OHT addresses that correspond to action forms which may not define (i,j), $1\le i\le NS$, $1\le j\le N$, $t=j,j+1,\ldots,N$.

Assertion 4.7 For the examination of MO(t), (i,j) is contained at all OHT addresses that correspond to action forms which may define (i,j), $1\le i\le NS$, $0\le j\le N$, $t=j,j+1,\ldots,N$.

Proof:
When (i,j) is defined at MO(j) and MO(j) $\ne$ UDF:

OHT(OP(j),sc(j,m),c(j)) := OHT(Op(j),sc(j,m),c(j))■(i,j)

for $m=1,2,\ldots,nsc(j)$;

thus, for the examination of MO(t) where $t = j$ and MO(j) $\ne$ UDF, (i,j) is contained at all OHT addresses that correspond to actions which may define (i,j), $1\le i\le NS$, $0\le j\le N$.

When (k,s) is defined at MO(s), MO(j) <=> MO(s), s > j, and MO(s) ≠ UDF:

```
For m=1,2,...,nsc(s)
    OHT(Op(s),sc(s,m),c(s)) := E
    where E := E ∪ (OHT(Op(s),sc(s,a),c(s)) ≠ E)
        for a=1,2,...,nsc(s).
```

From Theorem 4.1, sc(j) ∈ sc(s), and sc(j) ∉ sc(s) defines an additional set of source unit combinations which identify action forms that may define (i,j); and, since E contains all QMSUs at OHT(Op(s),sc(s),c(s)), (i,j) ∈ E. Therefore, for an examination of MO(t) where t=j+1,j+2,...,N and MO(t) ≠ UDF, (i,j) is contained at all OHT addresses that correspond to action forms which may define (i,j), 1≤i≤NS, 0≤j≤N.

From Assertions 4.4 and 4.6, it can be seen that for (i,j) defined at MO(t) where MO(t) = UDF, (i,j) is contained at all OHT addresses that correspond to action forms which may define (i,j), t=j,j+1,...,N; therefore, for the examination of MO(t), (i,j) is contained at all OHT addresses that correspond to action forms which may define (i,j), 1≤i≤NS, 0≤j≤N, t=j,j+1,...,N.

Q.E.D.

<u>Corollary</u> 4.7A(1) From Assertions 4.4, 4.6, and 4.7: (i,j) is contained at no OHT address that corresponds to an action form which may not define (i,j),1≤i≤NS, 0≤j≤N.

Corollary 4.7A(2) From Assertions 4.4, 4.6, and 4.7: OHT(a) contains a complete list of QMSUs that may be defined by the action form to which OHT(a) corresponds, $1 \le a \le nsc(i)$, $i=1,2,\ldots,N$.

Assertion 4.8 OHT[UHA(2,k)] contains a complete list of QMSUs which are equivalent to the current definition of unit {k}, $1 \le k \le NS$.

Proof: (assume UHA(1,k) = x)

Where (k,x) is defined at MO(x) and MO(x) ≠ UDF:

> UHA(2,k) := OHT'Op(x),sc(x,1),c(x)'.
>
> From Corollary 4.7A(2), OHT(Op(x),sc(x,1),c(x)) contains a complete list of QMSUs that may be defined by the action form to which OHT(Op(x),sc(x,1),c(x)) corresponds; therefore, for MO(x) ≠ UDF, OHT[UHA(2,k)] is complete, $1 \le k \le NS$.

When (k,x) is defined at MO(x) and MO(x) = UDF:

> UHA(2,k) := UHA(2,sc(x,1)).
>
> From Assertions 4.4, 4.6 and Corollary 4.7A(2), it can be seen that for MO(x) = UDF, OHT[UHA(2,k)] is complete, $1 \le k \le NS$.

Since OHT[UHA(2,k)] is complete for MO(x) = UDF and for MO(x) ≠ UDF, OHT[UHA(2,k)] contains a complete list of QMSUs which are equivalent to the current definition of unit {k}, $1 \le k \le NS$.

Q.E.D.

Assertion 4.9 For the examination of MO(i), all Type C and Type R alternate source units may be identified, i=1,2,...,N.

Proof:
From Assertion 4.8, all QMSUs equivalent to sc(i,1,j) are contained at OHT[UHA(2,sc(i,1,j))], j=1,2,...,nsc(i,1), i=1,2,...,N.
Q.E.D.

To show that Pass 1 identifies all Type P and Type S alternate sink units, the following assertions are made.

Assertion 4.10 For the examination of MO(i), all Type P alternate sink unit references may be identified, i=1,2,...,N.

Proof:
From Assertion 4.9, it can be seen that all equivalent action forms may be identified; and, from Corollaries 4.6A(2) and 4.7A(2), it can be seen that the index of all equivalent predecessor actions may be identified. Therefore, for the examination of MO(i), all Type P alternate sink unit references may be identified, i=1,2,...,N. (Assertion 4.10 is also immediate from Theorem 4.3)
Q.E.D.

Assertion 4.11 For the examination of MO(i), all Type S alternate sink unit references for MO(j) may be identified, i=2,3,...,N, j=1,2,...,N-1.

Proof:
From Assertion 4.10 and Theorem 4.2, it can be seen that for the examination of MO(i), all Type S alternate sink unit references for MO(j) can be identified, i=2,3,...,N, j=1,2,...,N-1.
Q.E.D.

Assertions 4.1 through 4.11 have been offered to show that Pass 1 identifies all Type C and Type R alternate source units and all Type P and Type S alternate sink units. The variables of form for actions are Type C, Type R, and Type A alternate source units and Type P and Type S alternate sink units. The following section offers proof that Pass 2 of the algorithm identifies all Type A alternate source units.

4.5.2 Pass 2: Proof

To show that Pass 2 identifies all Type A alternate source units, the following assertion is made (assume N actions in a program node and NS memory sub-units).

Assertion 4.12 If all Type P and Type S alternate sink units are identified at MO(i), then all Type A alternate source units may be identified, i=1,2,...,N.

Proof:

From def 4.4 and 4.5:

If QMSU (m,x) is an alternate sink unit for MO(k), then MO(x) <=> MO(k);

and, if (t,s) is an alternate sink unit for MO(k), then MO(k) <=> MO(s).

Therefore, sk(x) = m, MO(x) <=> MO(k), and MO(k) <=> MO(s); and, the conditions given in def 4.3 are satisfied for $1 \le m,t \le NS$, $1 \le x,k,s \le N$.

From def 4.3, 4.4, 4.5, and Corollary 4.3, it can be seen that if all Type P and Type S alternate sink units are identified at MO(i), then all Type A alternate source units can be identified, i=1,2,...,N.

Q.E.D.

The alternate unit identification algorithm has been described in detail and an algorithm proof has been offered. The next major section describes the relationship between the ALT, constructed by the algorithm, and machine data-flow fan-out capabilities.

4.6 ALT and Hardware Fan-out

The algorithm presented in this Chapter identifies all equivalent actions. These equivalent actions are represented at ALT(i,ask) for i=1,2,...,N. It can be seen that if MO(i) <=> MO(j), then sk(i) is a fan-out candidate at MO(j) and sk(j) is a fan-out candidate at MO(i) [52]; therefore, the information collected in ALT(i,ask) may be used to delete actions by the exploitation of parallel hardware fan-out capabilities. Also, the identification of equivalent actions establishes the basis for alternate action form identification.

4.7 Summary

This Chapter presented and proved the correctness of an algorithm which identifies all possible forms of each action in a program node. The ALT has been structured to contain the information which defines alternate action forms. In order for the algorithm to construct the ALT, alternate units were qualified as to type, and a hashing technique was proposed.

The algorithm described was coded using APL/360. The hashing function used for the program node given in Figure 3.1 was a residue modulo(213) arithmetic technique. There were 37 entries generated in the OHT (NS = 9).

With reference to the solution approach described in Section 3.3, this Chapter has presented an algorithm for the determination of all possible forms, F(m), of action(m), by the identification of equivalent actions and equivalent units, m=1,2,...,N. The next Chapter defines the minimal set of conditions required for an action to be a nonessential candidate for deletion.

CHAPTER V

DELETION CANDIDACY

5.1 Introduction

In Chapter 3, nonessential actions were defined in terms of particular states (0,1,2, or 3), within the unit assignment table (UA). In Chapter 4, an algorithm was presented which identifies all possible alternate action forms. The purpose of this Chapter is to describe a minimal set of conditions, within alternate forms of the program node, which must exist in order to delete an action as nonessential; i.e., alternate action forms are used to identify candidates for deletion.

The general concept displayed in this Chapter is as follows: Through the assignment of alternate source and sink units, UA states are determined which represent nonessential unit definitions. Section 5.2 describes the conditions required to change a particular state in the UA. Then, Section 5.3 shows how an action is deleted by the assignment of alternate source and sink units; i.e., conditions required for changing a set of states in the UA are defined. Included in Section 5.3, proofs for the sufficiency of conditions are offered.

It was found that order-isomorphisms may exist in deletion conditions. Section 5.4 describes the cases where these may occur and presents a basic theorem for the analysis of reciprocally dependent deletion conditions.

A simple example of deletion candidates is presented in Section 5.5. The example illustrates some of the concepts developed in this Chapter. From the deletion conditions described, it is possible to define deletion strategies for nonessential actions. These strategies will be discussed in Chapter 6.

5.2 Conditions Required to Change States in the UA

The purpose of this section is to describe those conditions which must exist, in order to change a state within the unit assignment table. In particular, states are to be changed so as to create a nonessential unit definition in the program node.

From an examination of the nonessential action definitions given in Section 3.3, it can be seen that the following state changes in the UA may be required for the deletion of an action.

1. State 1 changed to state 0: The deletion of a unit from its participation as a source unit for an action.

2. State 3 changed to state 2: Same as 1, above.

3. State 2 changed to state 0: The deletion of a unit from its participation as a sink unit for an action.

4. State 3 changed to state 1: Same as 3, above.

Those conditions which are required to make state changes 1 to 0 and 3 to 2 are as follows: The existence of a feasible alternate source unit; or, the deletion of an action from the program node. Those conditions which are necessary to make state changes 2 to 0 and 3 to 1 are as follows: The movement of a unit definition to another action; or, the deletion of an action from the program node. Section 5.3 defines the conditions required for an action to be deleted. The conditions of feasibility for an alternate source unit are described next.

5.2.1 Alternate Source Unit Assignment Feasibility

An alternate source unit, ALT(i,asc(j,k)), is feasible for assignment only if conditions exist which insure that the unit is equivalent at MO(i) to the unit in ALT(i,psc(j)), $1 \le i \le N$, $1 \le j \le nsc(i,1)$, $1 \le k \le nALT(i,asc(j))$. These required conditions are given in the definition below.

def. 5.1 Feasible alternate source unit: Unit {i} is a feasible alternate source unit for unit {t} at MO(x), $1 \le x \le N$, if, and only if, one of the following conditions is satisfied. [Assume unit {t} is defined at MO(s)]

1. $(t,s) \in$ ALT(x,psc(a)), $0 \le s \le N-1$, $1 \le a \le nsc(x,1)$;
   $(C,i,j) \in$ ALT(x,asc(a)), $0 \le j \le N-1$.

2. $(t,s) \in$ ALT(x,psc(a)), $0 \le s \le N-1$, $1 \le a \le nsc(x,1)$;
   $(R,i,j) \in$ ALT(x,asc(a)), $0 \le j \le N-2$;
   for $n=j+1,j+2,\ldots,x-1$

where UA(m,sk(j)) > 1,
MO(m) is a negated-forward candidate for deletion; or, MO(m) is a parallel-forward candidate for deletion with respect to MO(y), such that $y \geq x$.

3. $(t,s) \in$ ALT(x,psc(a)), $0 \leq s \leq N-1$, $1 \leq a \leq nsc(x,1)$;
(A,i,j,k) $\in$ ALT(x,asc(a)), $1 \leq j \leq N-1$, $1 \leq k \leq N-1$;
MO(k) is a parallel candidate for deletion with respect to MO(j);
for m=j+1,j+2,...,x-1
where UA(m,sk(k)) > 1,
MO(m) is a negated-forward candidate for deletion; or, MO(m) is a parallel-forward candidate for deletion with respect to MO(y), such that $y \geq x$.

The conditions for Type C, Type R, and Type A alternate source unit feasibility have been presented. The conditions required for the movement of a unit definition to another action are given in the section which describes parallel-forward and parallel-backward deletion conditions. Before the conditions for deletion candidacy are defined, it is necessary to describe condition conflicts; i.e., those cases where deletion conditions for an action may violate other deletion conditions for the same action.

5.2.2 Condition Conflicts

A basic requirement for the deletion of an action is that conditions for its deletion may not conflict with each other. For example, assume that MO(x) is examined as a possible negated-forward candidate for deletion. Further assume that its deletion requires UA(i,sk(x)) be changed from state 1 to state 0, and UA(j,sk(x)) be changed from state 3 to state 2. If the condition used to change the state in UA(i,sk(x)) is the assignment of alternate source unit (t,s), and the condition used to change the state in UA(j,sk(x)) requires the deletion of MO(s); then, the first condition requires the definition of unit {t} at MO(s), and the second condition requires the deletion of MO(s). Clearly, these two conditions are in conflict.

Simply stated, conditions are in conflict with each other, if each condition requires a different form of the same action (a deleted action is considered the null form of the action). It is important to note that conditions which define alternate source unit feasibility are also considered in the identification of conflicting conditions.

In order to identify a deletable action, it is necessary to define a minimal set of conditions for its deletion candidacy. Furthermore, these conditions must not conflict with each other. The next section describes the conditions for deletion candidacy.

5.3 Conditions for Deletion Candidacy

The purpose of this section is to define a minimal set of conditions which must exist, in order for an action to be deleted. Conditions for deletion candidacy are based upon an examination of each action independently; i.e., examination is given to each action as if it is the only action to be deleted from the program node. Of course, the deletion of one action is a dependent event, as it may affect the deletion status of other actions. The analysis of deletion dependencies among different deletion candidates is deferred to Chapter 6.

Along with each set of deletion conditions, a proof is offered. The proof shows that the conditions, when satisfied, are sufficient to identify all actions which conform to the nonessential action definitions given in Section 3.3. Before conditions and proofs are given, two original and basic theorems are presented. The intent of these theorems is to show the range of actions over which deletion examination is to be given.

5.3.1 Deletion Range Theorems

Theorem 5.1 Given a program node: An examination of all equivalent actions is sufficient to identify all parallel and redundant actions.

Proof:

From <u>def</u> 3.10 and 3.11, only equivalent actions may be parallel; and, from Theorem 3.1, the set of redundant actions is included in the set of parallel actions. Therefore, an examination of equivalent actions is sufficient to identify all parallel and redundant actions.

Q.E.D.

<u>Theorem</u> 5.2 Given a program node and NS memory sub-units: An examination of adjacent definitions of unit {k} is sufficient to identify all negated definitions of unit {k}, $1 \le k \le NS$.

Proof:

An examination of <u>def</u> 3.14 shows that MO(i) is negated-forward with respect to MO(j)

iff sk(i) = sk(j);

$i < j$;

$$\sum_{m=i+1}^{j} UA(m,sk(i)) = 2.$$

If MO(j) is negated-forward with respect to MO(t), then the MO(j) definition of sk(j) may be deleted; thus, UA(j,sk(j)) = 0. If MO(i) is negated-forward with respect to MO(j), and if MO(j) is negated-forward with respect to MO(t), then

$i < j < t$;

$sk(i) = sk(t)$; and

$$\sum_{m=i+1}^{t} UA(m,sk(i)) = 2.$$

Therefore, an examination of adjacent unit definitions is sufficient to identify all negated unit definitions.
Q.E.D.

It is intended that examination be given to each action in a first to last sequence within the program node. However, this order is not essential, since actions are examined independently. Deletion conditions are defined first, for parallel-forward candidacy; second, for parallel-backward candidacy; and third, for negated-forward candidacy.

5.3.2 Parallel-Forward Deletion Candidacy

MO(i) is a parallel-forward candidate for deletion with respect to MO(j) if, and only if, the following minimal set of conditions are satisfied and no condition conflicts occur, $1 \le i \le N-1$, $1 < j \le N$.

1. $(S,j) \in ALT(i,ask)$.

2. Either sk(i) may be fanned-out at MO(j),
   or sk(i) = sk(j),
   or MO(j) is a candidate for deletion and sk(i) can be used as a sink unit for Op(j).

3. For $m=i+1,i+2,\ldots,j$

   where $(sk(i),i) \in ALT(m,psc(r))$, $1 \le r \le nsc(m,1)$, at least one of the following conditions is satisfied:

   a) There exists a feasible alternate in $ALT(m,asc(r))$;

   b) $MO(m)$ is a negated-forward candidate for deletion;

   c) $MO(m)$ is a parallel-forward candidate for deletion;

   d) $MO(m)$ is a parallel-backward candidate for deletion.

4. For $m=i+1,i+2,\ldots,j-1$

   where $(sk(i),i) \in ALT(m,psk)$, at least one of the following conditions is satisfied:

   a) $MO(m)$ is a negated-forward candidate for deletion;

   b) $MO(m)$ is a parallel-forward candidate for deletion with respect to $MO(s)$, such that $s \ge j$;

   c) $MO(m)$ is a parallel-backward candidate for deletion with respect to $MO(s)$, such that $s < i$.

Proof of sufficiency:

Condition 1 states that (S,j) $\in$ ALT(i,ask). Therefore, from def. 4.5 and the alternate unit identification algorithm in Chapter 4, MO(i) <=> MO(j) and i < j.

Condition 2 insures that sk(i) may be fanned-out at MO(j), or sk(i) may replace sk(j) at MO(j). If sk(i) = sk(j), it is obvious that sk(i) may replace sk(j) at MO(j). If sk(i) $\neq$ sk(j), and sk(i) may not be fanned-out at MO(j), then sk(i) may replace sk(j) at MO(j) if, and only if, MO(j) is a deletion candidate. Also, sk(i) must be viable as a sink unit for Op(j). The viability requirement is necessary for processors whose "data-flow" sink units may differ from arithmetic sink units.

Condition 3 insures that if sk(i) is a prime source unit at MO(m) where $i < m \leq j$, it may be replaced by an alternate source unit; or, MO(m) may be deleted. If the alternate assignments are made at MO(m), or if MO(m) is deleted; then,

```
    for m=i+1,i+2,...,j
        UA(m,sk(i)) = 0|2.
```

Condition 4 states that where sk(i) is a prime sink unit at MO(m) for $i < m < j$, the MO(m) definition of sk(i) may be deleted or moved outside the range of deletion; therefore,

```
    for m=i+1,i+2,...,j-1
        UA(m,sk(i)) ≤ 1.
```

Condition 3 and Condition 4, together, yield the following parallel-forward deletion requirements:

$$\sum_{m=i+1}^{j-1} UA(m,sk(i)) = 0;$$

$$UA(j,sk(i)) = 0 \mid 2.$$

Q.E.D.

From Theorem 5.1, if there are t Type S alternate sink unit references in ALT(i,ask), then MO(i) is examined a maximum of t times for its deletion candidacy as a parallel-forward action. If MO(i) <=> MO(j) <=> MO(k), i < j < k, and MO(i) is not a parallel-forward candidate for deletion with respect to MO(j), then MO(i) cannot be a parallel-forward candidate for deletion with respect to MO(k). An exception to this is as follows: The only required and unsatisfied condition is that sk(i) replace sk(j). Conditions for parallel-backward candidacy are presented next.

5.3.3 Parallel-Backward Deletion Candidacy

MO(j) is a parallel-backward candidate for deletion with respect to MO(i) if, and only if, the following minimal set of conditions is satisfied and no condition conflicts occur, $1 \le i \le N-1$, $1 < j \le N$.

1. $(P,i) \in ALT(j,ask)$.

2. Either sk(j) may be fanned-out at MO(i),

   or sk(j) = sk(i),

   or MO(i) is a candidate for deletion and sk(j) can be used as a sink unit for Op(i).

3. For m=a+1,a+2,...,j [a = i for sk(j) ≠ sk(i); a = k of first UA(k,sk(j)) > 1 (k=i+1,i+2,...,j) for sk(j) = sk(i)],

   where (sk(j),j) ∈ ALT(m,psc(r)), 1≤r≤nsc(m,1),

   at least one of the following conditions is satisfied:

   a) There exists a feasible alternate in ALT(m,asc(r));

   b) MO(m) is a negated-forward candidate for deletion;

   c) MO(m) is a parallel-forward candidate for deletion;

   d) MO(m) is a parallel-backward candidate for deletion.

4. For m=i+1,i+2,...,j-1

   where (sk(j),j) ∈ ALT(m,psk),

   at least one of the following conditions is satisfied:

   a) MO(m) is a negated-forward candidate for deletion;

b) MO(m) is a parallel-forward candidate for deletion with respect to MO(s), such that $s \geq j$;

c) MO(m) is a parallel-backward candidate for deletion with respect to MO(s), such that $s < j$.

Proof of sufficiency:
Condition 1 and Condition 2, as with parallel-forward actions, insure that MO(i) <=> MO(j), $i < j$; and, either sk(j) may be fanned-out at MO(i), or sk(j) may replace sk(i) at MO(i).

Condition 3 for sk(j) ≠ sk(i) insures that if sk(j) is a prime source unit at MO(m) where $i < m \leq j$, it may be replaced by an alternate source unit; or, MO(m) may be deleted. Thus,

```
for m=i+1,i+2,...,j
    UA(m,sk(j)) = 0|2.
```

Condition 3 for sk(j) = sk(i) insures that if the MO(a) definition of sk(j) is used as a prime source unit at MO(m) where $i < a < m \leq j$, it may be replaced by an alternate source unit; or, MO(m) may be deleted. Thus, for $a = k$ of the first UA(k,sk(j)) > 1 (k=i+1,i+2,...,j),

```
for $m=i+1,i+2,...,a-1$
    $UA(m,sk(j)) \le 1;$
for $m=a+1,a+2,...,j$
    $UA(m,sk(j)) = 0|2;$
$UA(a,sk(j)) = 1|3.$
```

Condition 4 states that where sk(j) is a prime sink unit at MO(m) for $i < m < j$, the MO(m) definition of sk(j) may be deleted or moved outside the range of deletion. Therefore,

```
for $m=i+1,i+2,...,j-1$
    $UA(m,sk(j)) \le 1.$
```

Condition 3 and Condition 4, together, yield the following parallel-backward deletion requirement for $sk(j) \ne sk(i)$:

$$\sum_{m=i+1}^{j} UA(m,sk(j)) = 2;$$

and, for $sk(j) = sk(i)$:

```
For $m=i+1,i+2,...,j-1$
    $UA(m,sk(j)) \le 1.$
```

Q.E.D.

From Theorem 5.1, if there are t Type P alternate sink unit references in ALT(i,ask), then MO(i) is examined a maximum of t times for its deletion candidacy as a parallel-backward action. If MO(i) <=> MO(j) <=> MO(k), $i > j > k$, and MO(i) is not a parallel-backward candidate for deletion with respect to MO(j), then MO(i) cannot be a parallel-backward candidate for deletion with respect to MO(k). An exception to this is as follows: The only required and unsatisfied condition is that sk(i) replace sk(j). Conditions for negated-forward candidacy are presented next.

5.3.4 Negated-Forward Deletion Candidacy

MO(i) is a negated-forward candidate for deletion with respect to MO(j) if the following minimal set of conditions is satisfied and no condition conflicts occur, $1 \le i \le N$, $1 < j \le N+1$.

1. sk(i) = sk(j).

2. $i < j$.

3. For $m=i+1,i+2,\ldots,j$

   where sk(i) ∈ ALT(m,psc(r)), $1 \le r \le nsc(m,1)$,

   at least one of the following conditions is satisfied:

   a) There exists a feasible alternate in ALT(m,asc(r));

   b) MO(m) is a negated-forward candidate for deletion;

   c) MO(m) is a parallel-forward candidate for deletion;

   d) MO(m) is a parallel-backward candidate for deletion.

Proof of sufficiency:

Condition 3 states that if sk(i) is a prime source unit at MO(m) where $i < m \le j$, it may be replaced by an alternate source unit; or, MO(m) may be deleted. Therefore,

for $m=i+1,i+2,\ldots,j$

$UA(m,sk(i)) = 0|2;$

and, from Theorem 5.2 and Condition 3,

$$\sum_{m=i+1}^{j} UA(m,sk(i)) = 2.$$

Q.E.D.

From Theorem 5.2, the number of actions to be evaluated for negated-forward deletion candidacy can be determined by an examination of adjacent unit definitions in the UA. Simply stated, the number of actions evaluated equals the number of memory sub-unit redefinitions. This can be determined by an examination of $UA(i,j) > 1$, for $i=1,2,\ldots,NS$, while $j=1,2,\ldots,N+1$. Note that the figurative redefinition of units at UA(i,N+1) is included in the examination.

As stated previously, the conditions for deletion candidacy may contain conflicts. It was also found that conditions may contain order-isomorphisms. The next section will describe how this may occur, and will pose an original theorem which provides a basis for the analysis of order-isomorphic deletion conditions.

5.4 Reciprocally Dependent Deletion Conditions

The recursively defined conditions in the last section were proven to be sufficient for the identification of all possible deletion candidates. However, it can be shown that the conditions may never converge; i.e., analysis of deletion candidacy for an action may never terminate. This problem, called reciprocal dependency, is defined and examples are given; then, a theorem is posed which allows an analysis of reciprocal dependencies.

def. 5.2 Reciprocal Dependency: If during the analysis of MO(i) as a deletion candidate, it is found that MO(j) must be a deletion candidate; and then, during the analysis of MO(j) as a deletion candidate, it is found that MO(i) must be a deletion candidate, then MO(i) and MO(j) are reciprocally dependent with respect to deletion conditions. A more succinct definition is as follows: An ordered subset of deletion conditions for MO(i) is order-isomorphic with respect to an ordered subset of deletion conditions for MO(j), $1 \le i,j \le N$. For a general description of order-isomorphism, see Theorem 2.12 in [8].

To illustrate reciprocal dependency, the following program node is presented.

```
I(1):   ADD   {1;3}   {4}
I(2):   SUB   {1;4}   {5}
I(3):   ADD   {1;3}   {2}
I(4):   SUB   {1;2}   {3}
```

If unit {2} is not busy on exit (UA(5,2) = 2), then examination is given to MO(3) as a negated-forward candidate for deletion. If MO(3) is to be a negated-forward action, then at least one of the following conditions must exist:

a) (2,3) is replaced as a prime source unit at MO(4);

b) MO(4) is negated-forward;

c) MO(4) definition of unit {3} is moved.

An examination of the above program node shows that the MO(4) definition of unit {3} may be moved to MO(2); however, from Condition 3 for parallel-backward deletion, this movement requires the deletion of MO(3). Therefore, MO(3) and MO(4) are reciprocally dependent with respect to deletion conditions. A less elaborate example of reciprocal dependency is MO(1) and MO(2) in the following program node. (assume fan-out not possible)

```
I(1):    ADD    {1;2}    {3}
I(2):    ADD    {1;2}    {2}
```

A theorem is presented next which is basic to the analysis of reciprocal dependencies.

5.4.1 Reciprocal Dependency Analysis

Reciprocally dependent deletion conditions would present an unsolvable problem, if they were randomly isomorphic. However, it is possible to identify these conditions; furthermore, these conditions need not impede the finite analysis of deletion candidates. It is clear from def 5.2 that reciprocally dependent conditions may be identified.

The following theorem establishes a basis which allows a finite deletion candidacy analysis to be performed, even though reciprocal dependencies may occur.

Theorem 5.3 If MO(i) and MO(j) are reciprocally dependent with respect to deletion conditions, and if MO(j) satisfies all deletion conditions in a minimal set, except the condition which requires the deletion of MO(i); then, the deletion condition which requires the deletion of MO(j) for the deletion candidacy of MO(i) is satisfied, $1 \leq i,j \leq N$.

Proof:

Let C(I) and C(J) represent the minimal set of conditions for the deletion candidacy of MO(i) and MO(j), respectively. Also, let c(i) represent the condition which requires the deletion of MO(i), and let c(j) represent the condition which requires the deletion of MO(j). From def 5.2, $c(i) \in C(J)$, and $c(j) \in C(I)$.

If all C(J) are satisfied, except c(i), then c(j) has been reduced to c(i). This reduction yields the trivial statement: The deletion of MO(i) requires the deletion of MO(i). Therefore, c(j) is satisfied.

Q.E.D.

Theorem 5.3 is presented as proof that the deletion candidacy analysis will converge; i.e., all candidates for deletion can be identified by a finite analysis. Before consideration is given to deletion dependencies among actions, the deletion conditions are applied to a program node example.

5.5 Deletion Candidates: Example

An example is now presented to demonstrate the application of deletion conditions to a program node. The program node selected is given in Figure 3.1, with the unit assignment table given in Table 3.1, and the alternate unit assignment table given in Table 4.1.

Before conditions are applied, examination is given to the UA, in order to determine those actions which are to be tested for negated-forward deletion candidacy. Also, in order to determine actions which are to be tested for parallel deletion candidacy, examination is given to ALT(i,ask(j)), j=1,2,...,nALT(i,ask), i=1,2,...,N.

Based upon Theorem 5.2, an examination of the UA given in Table 3.1 yields the following actions which are to be tested for negated-forward deletion candidacy. (MO(N+1) represents the "busy on exit" status of a memory sub-unit)

MO(4) with respect to MO(N+1)

MO(6) with respect to MO(N+1)

MO(1) with respect to MO(8)

MO(2) with respect to MO(N+1)

MO(3) with respect to MO(N+1)

MO(7) with respect to MO(N+1)

Based upon Theorem 5.1, an examination of the ALT(i,ask) entries yields the following actions to be tested for parallel-forward deletion candidacy, i=1,2,...,N.

MO(3) with respect to MO(6)

MO(5) with respect to MO(7)

Also, based upon Theorem 5.1, an examination of the ALT(i,ask) entries yields the following actions to be tested for parallel-backward deletion candidacy.

MO(6) with respect to MO(3)

MO(7) with respect to MO(5)

The objective is to identify a minimal set of conditions which are satisfied and which may be used to effect the deletion of the action tested. It is clear, from the deletion conditions, that fan-out capabilities may affect deletion candidacy. Specifically, without fan-out capabilities, the parallel deletion candidacy of an action requires the deletion of the sink unit definition at the action to which it is parallel; exception: A redundancy case where sink units are the same. In order to accommodate this, deletion candidates will be determined with and without fan-out capabilities.

It is important to note that conditions are to be applied in the order of their appearance in Section 5.3; furthermore, once a minimal set of conditions is satisfied, the analysis is terminated, and the action under examination becomes a deletion candidate, Of course, more than one minimal set of conditions may be satisfied; i.e., different deletion strategies may exist for an action.

5.5.1 Fan-out Possible

Fan-out capabilities are assumed to be possible at each parallel action. Deletion candidates are described as to type (negated-forward, parallel-forward, and parallel-backward), and the minimal set of conditions satisfied is described. Only conditions which require a specific action form are described. (acronym "wrt" is used for "with respect to")

Negated-forward candidates for deletion:

Action	wrt Action	Conditions
MO(6)	MO(N+1)	1. No alternate action form conditions are required; unit {5} is not used as a source unit in successor actions.
MO(2)	MO(N+1)	1. Unit {6} may be replaced at MO(3) by unit {4}, because (C,4,0) is feasible. 2. Unit {6} may be replaced at MO(7) by unit {4}, because (C,4,0) is feasible.

MO(3)	MO(N+1)	1. Unit {7} may be replaced at MO(6) by unit {5}, because (A,5,3,6) is feasible. 2. Unit {7} may be replaced at MO(8) by unit {5}, because (A,5,3,6) is feasible.
MO(7)	MO(N+1)	1. No alternate action form conditions are required; unit {9} is not used as a source unit in successor actions.

Parallel-forward candidates for deletion:

<u>Action</u> <u>wrt</u>	<u>Action</u>	<u>Conditions</u>
MO(3)	MO(6)	1. Unit {7} may be replaced at MO(6) by unit {5}, because (A,5,3,6) is feasible.
MO(5)	MO(7)	1. No alternate action form conditions are required; unit {8} is not used as a source unit in MO(6) through MO(7).

Parallel-backward candidates for deletion:

Action	wrt Action	Conditions
MO(6)	MO(3)	1. No alternate action form conditions are required; unit {5} is not used as a source unit in MO(3) through MO(6).
MO(7)	MO(5)	1. No alternate action form conditions are required; unit {9} is not used as a source unit in MO(6) through MO(7).

The set of deletion candidates have been determined for the case where fan-out is possible. The next section describes the deletion candidates and conditions when fan-out is not possible.

5.5.2 Fan-out Not Possible

The minimal set of conditions for negated-forward candidates for deletion are the same as those where fan-out is possible. However, the parallel candidates are affected by fan-out capabilities, in this example.

Parallel-forward candidates for deletion:

Action	wrt Action	Conditions
MO(3)	MO(6)	1. Unit {7} may be replaced at MO(6) by unit {5}, because {A,5,3,6} is feasible. 2. MO(6) definition of unit {5} may be parallel-backward with respect to MO(3) (note order-isomorphism).
MO(5)	MO(7)	1. MO(7) definition of unit {9} may be negated forward with respect to MO(N+1).

Parallel-backward candidates for deletion:

Action	wrt Action	Conditions
MO(6)	MO(3)	1. MO(3) definition of unit {7} may be negated-forward with respect to MO(N+1).

The example has been presented to demonstrate the application of deletion conditions. It is by no means supportive of the condition sufficiencies. Sufficiency proofs have been offered in Section 5.3.

5.6 Summary

This Chapter has been presented to define those conditions which are required for the determination of deletion candidates. The general theoretical approach has been to define conditions which are necessary to effect changes in UA states. These new states are intended to conform to the nonessential action definitions given in Section 3.3.

An important concept which is presented in this Chapter is reciprocal dependency. A theorem has been posed which may be used in the analysis of order-isomorphic deletion conditions.

With reference to the solution approach described in Section 3.3, Chapter 4 presented an algorithm for the identification of all forms, F(m), of action(m), m=1,2,...,N. Through an analysis of F(m), this Chapter has defined the conditions necessary to identify all deletion candidates, D. It is now necessary to describe the deletion strategies which may be employed in the deletion of actions represented in D. Chapter 6 describes deletion strategies and strategy conflicts, so that criteria of optimal action deletion may be defined.

CHAPTER VI

ANALYSIS OF DELETION CANDIDATES

6.1 Introduction

Chapter 4 showed that all possible alternate action forms can be identified. This alternate action form information is used, in Chapter 5, to develop conditions required for an action to be a nonessential candidate for deletion. The purpose of this Chapter is to describe the analysis of deletion candidates.

As described previously, the deletion candidates are identified independently. In order to effect action deletions, it is necessary to evaluate the dependencies among deletion candidates. The objective of deletion candidate analysis is to determine the greatest number of actions which can be deleted from a program node. Section 6.2 describes the variables of deletion candidate analysis.

In a broad perspective, deletion strategies are identified for each deletion candidate. Then, through an analysis of deletion strategies and strategy conflicts, an optimal solution is determined. Section 6.3 describes the strategies and conflicts, and Section 6.4 presents a means for their representation.

The graph-theoretic approach given in Chapter 3 describes the combinatoric nature of this particular optimization problem. Simply stated, analysis is performed to find the combination of deletion strategies which results in the greatest number of action deletions. Section 6.5 discusses criteria of optimal strategy selection and presents a strategy selection approach. The approach presented is not designed for its efficiency in terms of order of computation; however, this is not the purpose of this study. This study is designed to show that alternate action forms are determinable; and, deletion strategies may be defined which include an analysis of these forms.

6.2 General Analysis Approach

The purpose of this section is to describe the general approach used to evaluate deletion candidates. The analysis of deletion candidates involves an evaluation of the following three variables.

1. Deletion strategies: Those strategies which may be employed in the deletion of an action.

2. Strategy locations: Those actions in the program node where a strategy must be applied, in order to delete an action.

3. Strategy conflicts: Those strategies which cannot be used together for the deletion of an action or a set of actions.

As described in Chapter 5, an action is a deletion candidate if, and only if, a minimal set of deletion conditions is satisfied. Alternate source unit assignment feasibility conditions are included in the set of deletion conditions.

For the purpose of deletion analysis, deletion conditions are placed in two broad categories: Constraint conditions and action form related conditions. Constraint conditions describe an equivalence or positional relationship which is basic to an action's deletion candidacy or to a unit's assignment feasibility; e.g., MO(i) <=> MO(j) and i < j. Clearly, no strategy is implied by constraint conditions.

Action form related conditions require the existence of a specific action form; e.g., alternate source unit assignment or negation of some action. Deletion strategies are determined from an examination of action form related conditions. More precisely, deletion strategies are action form related conditions.

In order to use the deletion strategy variable in candidate analysis, it is necessary to identify where, in the program node, these strategies are to be applied; i.e., the strategy locations. Also, it is necessary to determine the strategies available for application at these different locations. For example, assume that a prime source unit reference must be changed at MO(x), in order to delete MO(y). The prime source unit field of MO(x) is a location which requires some strategy. The strategies available at this location may, for instance, be source unit replacement or the negated-forward deletion of MO(x).

The identification of the deletion strategies at strategy locations is sufficient to determine the ways in which a candidate may be deleted. The third variable of analysis, strategy conflicts, is necessary to evaluate which set of actions may be deleted from the program node.

In summary, the general approach used to evaluate deletion candidates is to first, identify all possible deletion strategies; second, determine at which locations strategies are required; and third, evaluate conflicting strategies. The next section presents a detailed description of the three analysis variables.

6.3 Analysis Variables

The variables of deletion candidate analysis are strategies, strategy locations, and strategy conflicts. This section describes these variables and how they are determined. The next major section presents a means for the representation of these variables.

Deletion strategies are described in terms of condition options which may exist for the deletion of an action. Strategy locations (points) are described with reference to each type of deletion condition. Strategy conflicts are explained in terms of strategy options which may not be used together, to delete an action or set of actions.

6.3.1 Strategy Options

Strategies which may be employed in the deletion of an action are determined from an examination of action form related deletion conditions. These strategies are divided into two general classes: Those which involve the replacement of prime source units; and, those which involve the deletion of an action. These two classes are the options available for the satisfaction of nonconstraint type deletion conditions. The second class, action deletion, may be performed by the application of five possible strategies. Thus, there are six strategy options available for the deletion of an action. These options are described below.

1. Source unit replacement (SRi): Those strategies which require the assignment of a feasible alternate source unit as a prime source unit; i is the source unit referenced (e.g., i = 2 means replacement of the second source unit).

2. Negated-forward deletion (NF): Those strategies which require the negated-forward deletion of an action.

3. Parallel-forward deletion with fan-out (PFF): Those strategies which require the parallel-forward deletion of an action, where sink unit fan-out is performed.

4. Parallel-backward deletion with fan-out (PBF): Those strategies which require the parallel-backward deletion of an action, where sink unit fan-out is performed.

5. Parallel-forward deletion without fan-out (PFR): Those strategies which require the parallel-forward deletion of an action, where nontrivial sink unit replacement is performed. (recall, sk(i) = sk(j) is the trivial case)

6. Parallel-backward deletion without fan-out (PBR): Those strategies which require the parallel-backward deletion of an action, where nontrivial sink unit replacement is performed.

The deletion of an action may be described completely in terms of these strategy options. Specifically, the deletion of an action may be expressed as a combination of source unit replacements, negated actions, and parallel actions. From this, it can be seen that the set of deletion candidates together with all feasible alternate source units completely define the set of deletion strategy options.

The deletion of an action is nonunique, as there may exist more than one set of strategies for its deletion. Therefore, it is necessary to identify all possible sets of strategies which may be employed in the deletion of an action. This is accomplished by the identification of actions where strategies are to be applied; then, all strategy options are determined. The next section discusses the identification of actions (strategy points) where strategies are to be applied; and, describes the determination of strategy options at these strategy points.

6.3.2 Strategy Points

Many strategy sets may exist for the deletion of an action. The concept of strategy points is helpful in the identification of these sets. This section describes the strategy point variable of analysis. Also, a theorem is presented which is basic to the representation and analysis of deletion strategies. First, strategy point is formally defined.

def. 6.1 Strategy point: Any action's source or sink unit which requires alteration (replacement, deletion, or movement), in order to delete an action as nonessential.

Strategy points are referenced by MO(i.j), where i is the index of the action which contains a strategy point, and j is an identifier of the unit within MO(i) which requires alteration. To clarify, if nsc(i,1) = 2 and nsk(i) = 1, then MO(i.2) references the second source unit, and MO(i.3) references the sink unit. As a more descriptive example, the following program node is presented (note that at MO(4), units {3}, {4}, and {5} are equivalent).

```
I(1):   ADD    {2;3}   {5}
I(2):   GATE    {5}    {4}
I(3):   ADD    {2;3}   {3}
I(4):   SUB    {3;6}   {7}
I(5):   ADD    {6;7}   {3}
```

If MO(3) is to be a negated-forward deletion candidate with respect to MO(5), it can be seen that the MO(4) reference to prime source unit {3} must be replaced or deleted. Thus, MO(4.1) is a strategy point for the negated-forward deletion of MO(3) with respect to MO(5). It is necessary to determine all strategy options which may be employed at this strategy point. If unit {7} is busy on exit (MO(4.1) cannot be deleted), then the following strategies are available for the replacement of unit {3} at strategy point MO(4): Replacement by unit {5}, because (C,5,1) is feasible; and, replacement by unit {4}, because (C,4,2) is feasible.

The conditions described in Chapter 5 are applied to each strategy point, in order to identify all strategy options. Based upon those action form related conditions, strategy points are now described for negated-forward, parallel-forward, and parallel-backward candidates for deletion.

If MO(i) is examined for negated-forward deletion candidacy with respect to MO(j), then i < j, and sk(i) = sk(j). The strategy points for its deletion are given by the following:

```
For m=i+1,i+2,...,j
    where UA(m,sk(i)) = 1|3.
```

If MO(i) is examined for parallel-forward deletion candidacy with respect to MO(j), then i < j, and MO(i) <=> MO(j). The strategy points for its deletion are given by the following: (assume MO(j.3) references sink unit)

```
For m=i+1,i+2,...,j
    where UA(m,sk(i)) = 1|3;
for m=i+1,i+2,...,j-1
    where UA(m,sk(i)) = 2|3; and,
if fan-out is not possible at MO(j),
    MO(j.3).
```

If MO(j) is examined for parallel-backward deletion candidacy with respect to MO(i), then i < j, and MO(j) <=> MO(i). The strategy points for its deletion are given by the following [if sk(j) = sk(i), then a = k of first UA(k,sk(j)) > 1 (k=i+1,i+2,...,j); if sk(j) ≠ sk(i), then a = i]:

```
For m=a+1,a+2,...,j
    where UA(m,sk(j)) = 1|3;
```

for m=i+1,i+2,...,j-1
where UA(m,sk(j)) = 2|3; and,
if fan-out is not possible at MO(i),
MO(i.3).

It is clear that if all strategy options are determined for each strategy point, then all possible combinations of strategies are identified for each deletion candidate. The representation and analysis of strategy combinations is facilitated by the identification of unique strategies. The following original theorem is presented as a basis upon which strategy combinations may be represented and evaluated.

Theorem 6.1 Given i equivalent actions in the program node PN, j feasible alternate source units over PN, and k redefined sink units in PN (units not busy on exit are designated as unit definitions); then, the number of unique strategies available for nonessential action deletion over PN is bounded above by 2i + j + k.

Proof:
The deletion conditions which relate to action forms are forward parallelism and negation, backward parallelism, and alternate source unit assignment. From these deletion conditions, Theorem 6.1 is immediate; since equivalent actions define parallel actions, and redefinitions include the busy on exit status of memory sub-units. Finally, if fan-out is possible at each equivalent action, then each

parallel candidate for deletion may be effected by sink unit fan-out or by sink unit replacement (2i).

Q.E.D.

Corollary 6.1 From Theorem 6.1 and Theorem 5.2:
The number of deletion strategy combinations is finite, and deletion strategies are convergent.

Two of the three variables of analysis, deletion strategies and strategy points, have been described. Also, their determination and relationship have been discussed. Finally, deletion strategies have been shown to be finite and convergent. The analysis of deletion candidates is described in terms of strategy options available at strategy points. The third variable required for the analysis is strategy conflicts. Strategy conflicts provide a means of evaluating the relationship between deletion candidates.

6.3.3 Strategy Conflicts

The identification of strategy options at strategy points provides a means for determining ways in which a deletion candidate may be deleted from the program node. Of course, in order to delete the maximum number of candidates, there must exist some means whereby sets of deletable actions may be identified and evaluated; i.e., examination is given to determine which set of candidates may be deleted from the program node. Relationships among deletion candidates are evaluated by strategy conflict analysis.

For the purpose of analysis, strategy conflicts are divided into two categories: Absolute strategy conflicts and deletion impedance conflicts. The following definitions are presented to qualify these categories; then, an example of each category is presented.

def. 6.2 Absolute strategy conflicts: Those strategy pairs which imply different deletion approaches of the same unit definition (null deletion approach included); and, those strategies which describe different alternate source unit assignments for the same prime source unit of an action.

def. 6.3 Deletion impedance conflicts: Those strategy pairs in which one strategy results in the null form of MO(i), while the other strategy requires the nonnull form of Op(i), sc(i,j), and c(i), $1 \le i \le N$, $1 \le j \le nsc(i)$

As an example of the first category, absolute strategy conflicts, consider the following case: (C,4,3) is a feasible alternate source unit at MO(8), and MO(3) is a negated-forward candidate for deletion with respect to MO(9). Two strategies are implied by this case: 1. The negated-forward deletion of the MO(3) definition of unit {4}; and, 2. The assignment of unit {4}, defined at MO(3), as a prime source unit. Strategy 1 requires a negated-forward deletion approach for the unit {4} definition at MO(3), while Strategy 2 requires a null deletion approach for the unit {4} definition at MO(3); therefore, from def 6.2, Strategy 1 and Strategy 2 are absolute conflicts.

As an example of the second category, deletion impedance conflicts, consider the following case: MO(3) is a negated-forward candidate for deletion with respect to MO(9), and MO(8) is a parallel-backward candidate for deletion with respect to MO(3). Two strategies are implied by this case: 1. The negated-forward deletion of MO(3); and, 2. The parallel-backward deletion of MO(8). The first strategy results in the null form of MO(3), while the second strategy requires the existence of Op(3), sc(3,j), and c(3), $1 \le j \le nsc(3)$. Therefore, from def 6.3, Strategy 2 impedes the action deletion described by Strategy 1. It should be noted that deletion impedance conflicts may exist in the optimal strategy selection. However, absolute strategy conflicts may never exist in any selected strategy set.

It is clear from the conflict definitions that if all strategy options are identified, then all conflicts may be determined. Strategy options, strategy points, and strategy conflicts have been discussed. It is intended that these three variables of analysis be applied, in order to determine a set of strategies which will optimize the objective function. The optimal strategy selection criteria are discussed in Section 6.5. However, before strategy selection is examined, a means of representing the analysis variables is presented.

6.4 Representation of Analysis Variables

The analysis of deletion candidates is facilitated by the representation of strategy options, strategy points, and strategy conflicts. The proposed scheme is developed as a means of concisely representing the following relationships among analysis variables.

1. Different action deletion approaches; i.e., the different ways in which an action may be deleted.

2. Strategy points defined for different action deletion approaches.

3. Absolute strategy conflicts among strategy options.

4. Deletion impedance conflicts among strategy options.

5. Strategy options available for the alteration of strategy points.

In a broad perspective, these relationships are to be represented in a form which will lend itself to deletion candidate analysis. Specifically, the representation form should contain all the information necessary to select an optimal strategy set. The selection of strategies is deferred to Section 6.5. The representation form, strategy table (ST), is described next.

6.4.1 Strategy Table (ST)

All of the analysis relationships may be represented independently, in tabular form. This section describes the strategy table (ST) whose purpose is the representation of these relationships. First, the structure of the strategy table is described; then, an example is presented.

6.4.1.1 ST Structure

It is clear that all deletion candidates are represented in the set of all possible strategy options. Furthermore, deletable actions (deletion candidates) and feasible alternate source unit assignments make up the set of all possible strategy options for a program node. The ST is structured to represent all strategy options and strategy points. Therefore, all deletion candidates, feasible alternate source unit assignments, and locations which require a strategy are represented.

Strategy points are identified by an examination of the UA and the ALT. Strategy options which may be applied to these points are determined by the application of action form related deletion conditions. These conditions include alternate source unit assignment feasibility conditions.

To describe the structure, the following notation is presented.

nD elements in the set of deletion candidates D.

d(i) unique strategy points defined for the deletion of the action represented in D(i), $1 \le i \le nD$.

nF elements in the set of unique feasible alternate source units F, [F is determined by those possible assignments at d(i)].

f(j) unique strategy points defined for the feasibility of the alternate source unit represented in F(j), $1 \le j \le nF$.

Using this notation, the ST is defined to be a K by K matrix where

$$K = nD + nF + \sum_{i=1}^{nD} d(i) + \sum_{j=1}^{nF} f(j).$$

From the above equation, it can be seen that the ST is structured to represent the different deletion candidacy options available for an action's deletion (nD); the different feasible alternate source units (nF); the strategy points for an action's deletion [d(i)]; and, the strategy points for a unit's alternate source assignment feasibility [f(j)]. Thus, all possible strategies and strategy points are represented in the ST.

The strategy table is divided into three sub-matrices. Each sub-matrix is designed to represent one or more of the relationships described previously. The purpose, organization, and elements of each sub-matrix are now described.

1. Strategy conflicts sub-matrix (SC): The purpose of this sub-matrix is to represent the conflict relationships among strategy options; also, by its organization, the SC represents the different action deletion approaches. Using the notation described in this section, the SC is an (nD+nF) by (nD+nF) matrix. ST(i,j) is resident in SC, if $1 \le i \le (nD+nF)$, and $1 \le j \le (nD+nF)$. Furthermore, the SC is organized to represent deletion candidates in ascending action index sequence, with all action deletion approaches grouped by action. ST(1,1) through ST(nD,nD) represent deletion candidates. ST(nD+1,nD+1) through ST(nD+nF,nD+nF) represent alternate source unit assignments. Therefore, the SC represents all possible strategy options. Each element in the SC sub-matrix assumes one of the following values:

SC(i,j) = 0, if strategy option i does not conflict with strategy option j.

SC(i,j) = 1, if strategy option i is an absolute strategy conflict with respect to strategy j.

SC(i,j) = 2, if strategy option i impedes the action deletion described by strategy option j.

SC(i,j) = 3, if strategy option j impedes the action deletion described by strategy option i.

SC(i,j) = 4, if strategy option i impedes the action deletion described by strategy option j, and strategy option j impedes the action deletion described by strategy option i.

Since no action deletion is identified by feasible alternate source unit assignments, $SC(i,j) = 0|1|2$ for $nD < i \leq (nD+nF)$ and $1 \leq j \leq nD$; $SC(i,j) = 0|1|3$ for $1 \leq i \leq nD$ and $nD < j \leq (nD+nF)$; and, $SC(i,j) = 0|1$ for $nD < i \leq (nD+nF)$ and $nD < j \leq (nD+nF)$.

2. Strategy points for strategy options sub-matrix (PO): The purpose of this sub-matrix is to represent those strategy points which require alteration, in order for a strategy option to exist. The PO is a $(nD+nF)$ by $(K-(nD+nF))$ matrix. $ST(i,j)$ is resident in PO, if $1 \leq i \leq (nD+nF)$, and $(nD+nF) < j \leq K$. Each element in the PO sub-matrix assumes one of the following values:

$PO(i,j) = 0$, if strategy option i does not require alteration of strategy point j.

$PO(i,j) = 1$, if strategy option i requires alteration of strategy point j.

3. Strategy options at strategy points sub-matrix (SP): The purpose of this sub-matrix is to represent those strategy options available for the alteration of a strategy point. The SP is a $(K-(nD+nF))$ by $(nD+nF)$ matrix. $ST(i,j)$ is resident in SP, if $(nD+nF) < i \leq K$, and $1 \leq j \leq (nD+nF)$. Each element in the SP sub-matrix assumes one of the following values:

$SP(i,j) = 0$, if strategy option j may not be used to alter strategy point i.

SP(i,j) = 1, if strategy option j may be used to alter strategy point i.

It is noted that although the ST is structured as a K by K array, ST(nD+nF+1,nD+nF+1) through ST(K,K) are unused elements. A review of the ST structure shows that all analysis relationships are represented. The next section develops the representation of the program node given in Figure 3.1 into a strategy table form.

6.4.1.2 ST Example

Chapter 5 showed the deletion candidates for the program node given in Figure 3.1. Therefore, the deletion candidates subset of strategy options has been identified. The successive application of the conditions described in Chapter 5 will yield the remaining strategy options; namely, the feasible alternate source unit assignments. The strategy options and related strategy points are given in Table 6.1 (acronym "wrt" is used for "with respect to"). The ST ID entries in Table 6.1 and Table 6.2 are reference numbers which correspond to entries in the strategy table. For generality, fan-out is assumed to be possible at all equivalent actions.

Table 6.1 Strategy Options and Strategy Points for Program Node Given in Figure 3.1

ST ID.	Action or ALT(asc)	Strategy Option	wrt Action	Strategy Points	ST refs.
1	MO(2)	NF	MO(N+1)	MO(3.2) MO(7.2)	18,22
2	MO(3)	NF	MO(N+1)	MO(6.1) MO(8.1)	20,24
3	MO(3)	PFF	MO(6)	MO(6.1)	20
4	MO(3)	PFR	MO(6)	MO(6.1) MO(6.3)	20,21
5	MO(5)	PFF	MO(7)		
6	MO(5)	PFR	MO(7)	MO(7.3)	23
7	MO(6)	NF	MO(N+1)		
8	MO(6)	PBF	MO(3)		
9	MO(6)	PBR	MO(3)	MO(3.3)	19
10	MO(7)	NF	MO(N+1)		
11	MO(7)	PBF	MO(5)		
12	(C,4,0)	SR2	MO(3)		
13	(C,4,0)	SR2	MO(7)		
14	(A,5,3,6)	SR1	MO(6)	MO(6.3)	21
15	(A,7,6,3)	SR1	MO(8)	MO(3.3)	19
16	(A,5,3,6)	SR1	MO(8)	MO(6.3)	21
17	(C,5,6)	SR1	MO(8)		

The application of deletion and alternate source unit feasibility conditions to the strategy points given in Table 6.1 yields the strategy options which may be employed in the alteration of strategy points. Table 6.2 shows the strategy options at strategy points. The information shown in Tables 6.1 and 6.2 is used in the construction of the strategy table given in Table 6.3.

Table 6.2 Strategy Options at Strategy Points for Program Node Given in Figure 3.1

		STRATEGY OPTIONS			
ST ID.	Strategy Points	Action or ALT(asc)	Strategy Option	wrt Action	ST refs.
18	MO(3.2)	(C,4,0)	SR2	MO(3)	12
"	" "	MO(3)	NF	MO(N+1)	2
"	" "	MO(3)	PFF	MO(6)	3
"	" "	MO(3)	PFR	MO(6)	4
19	MO(3.3)	MO(3)	NF	MO(N+1)	2
"	" "	MO(3)	PFF	MO(6)	3
"	" "	MO(3)	PFR	MO(6)	4
20	MO(6.1)	(A,5,3,6)	SR1	MO(6)	14
"	" "	MO(6)	NF	MO(N+1)	7
"	" "	MO(6)	PBF	MO(3)	8
"	" "	MO(6)	PBR	MO(3)	9
21	MO(6.3)	MO(6)	NF	MO(N+1)	7
"	" "	MO(6)	PBF	MO(3)	8
"	" "	MO(6)	PBR	MO(3)	9
22	MO(7.2)	(C,4,0)	SR2	MO(7)	13
"	" "	MO(7)	NF	MO(N+1)	10
"	" "	MO(7)	PBF	MO(5)	11
23	MO(7.3)	MO(7)	NF	MO(N+1)	10
"	" "	MO(7)	PBF	MO(5)	11
24	MO(8.1)	(A,7,6,3)	SR1	MO(8)	15
"	" "	(A,5,3,6)	SR1	MO(8)	16
"	" "	(C,5,6)	SR1	MO(8)	17

Table 6.3 Strategy Table for Program Node in Figure 3.1

				STRATEGY OPTIONS																	STRATEGY POINTS						
				deletion candidates											ALT unit assignments												
				2	3	3	3	5	5	6	6	6	7	7													
				1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
STRATEGY OPTIONS	deletion candidates	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0
		3	2	0	0	1	1	0	0	0	3	3	0	0	3	0	3	1	3	0	0	0	1	0	0	0	1
		3	3	0	1	0	1	0	0	2	4	4	0	0	3	0	3	0	3	0	0	0	1	0	0	0	0
		3	4	0	1	1	0	0	0	2	4	4	0	0	3	0	3	0	3	0	0	0	1	1	0	0	0
		5	5	0	0	0	0	0	1	0	0	0	2	4	0	0	0	0	0	0	0	0	0	0	0	0	0
		5	6	0	0	0	0	1	0	0	0	0	2	4	0	0	0	0	0	0	0	0	0	0	0	1	0
		6	7	0	0	3	3	0	0	0	1	1	0	0	0	0	1	3	1	1	0	0	0	0	0	0	0
		6	8	0	2	4	4	0	0	1	0	1	0	0	0	0	0	3	0	1	0	0	0	0	0	0	0
		6	9	0	2	4	4	0	0	1	1	0	0	0	0	0	0	3	0	1	0	1	0	0	0	0	0
		7	10	0	0	0	0	3	3	0	0	0	0	1	0	3	0	0	0	0	0	0	0	0	0	0	0
		7	11	0	0	0	0	4	4	0	0	0	1	0	0	3	0	0	0	0	0	0	0	0	0	0	0
	ALT assignments		12	0	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
			13	0	0	0	0	0	0	0	0	0	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0
			14	0	2	2	2	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0
			15	0	1	0	0	0	0	2	2	2	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0
			16	0	2	2	2	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	1	0	0	0
			17	0	0	0	0	0	0	1	0	1	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0
STRATEGY POINTS			18	0	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0							
			19	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0							
			20	0	0	0	0	0	0	1	1	1	0	0	0	0	1	0	0	0							
			21	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0							
			22	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0							
			23	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0							
			24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1							

The variables of analysis were described in Section 6.3. The representation of these variables, along with an example, has been presented in this section. In the next section, these variables are used to show that the selection of an optimal strategy set is possible.

6.5 Optimal Strategy Selection Criteria

Alternate and equivalent action forms are identified by a deterministic algorithm. These action forms are used in the definition of a minimal set of deletion conditions. The deletion conditions are applied to actions in a program node, in order to determine those actions which are nonessential candidates for deletion. Examination is given to these deletion candidates, to determine all strategies which may be employed in their deletion. The inclusion of alternate action forms in the deletion of nonessential actions requires one additional step: The selection of a set of strategies which results in the maximum number of deleted actions.

From the previous sections in this Chapter, it is clear that all deletion strategies, strategy points, and strategy conflicts can be determined. Furthermore, it is asserted that the information contained in the strategy table is sufficient to identify those strategies which may be employed in the deletion of an action or a set of actions. The strategy table is used throughout this section to aid in the description of a strategy selection approach.

First, the determination of an optimal strategy set is proved possible; then, an example is presented. The optimal strategy selection approach is presented with the objective of showing feasibility. The implementation technique used for such an approach is not evaluated in this study.

6.5.1 Optimal Strategy Selection Approach

The objective function may be restated as the selection of a set of strategy options which results in the maximum number of action deletions. The following original theorems establish a basis for the description of a feasible optimal strategy selection approach.

Theorem 6.2 Given a program node PN: Let D be all strategy options which describe action deletions. If all absolute strategy conflicts and deletion impedance conflicts are identified for strategies in D; then, an optimal solution upper bound can be determined, and all strategy option sets in D which possibly yield the optimal solution can be determined.

Proof:
This theorem is immediate, since absolute conflicts identify all pairs of strategy options which can be used together; and, deletion impedance conflicts identify all pairs of strategy options which result in the impedance of either or both action deletions described by the pair of strategy options. From def 6.2 and 6.3, no strategy option pair in D

may represent both an absolute and an impedance conflict; and, absolute conflicts can occur in D only for strategy options which describe the deletion of the same action. If D contains all strategy options which describe action deletions, then all deletion candidates are represented in D. Therefore, an exhaustive examination of strategy option pairs which are not absolute conflicts will identify all possible combinations of unique action deletions from PN. An exhaustive examination of impedance conflicts within these possible combinations will identify a maximum number of action deletions (optimal solution upper bound); and of course, all possible combinations (strategy option sets in D) which can yield an optimal solution are also identified. Q.E.D.

<u>Theorem</u> 6.3 Given a program node PN: Let D be all strategy options which describe action deletions; F be all strategy options which describe feasible alternate source unit assignments; and, P be all unique strategy points. With no consideration given to F and P: Let G be a set of s strategy options, $G \subset D$, with no absolute strategy conflicts and with t impeded action deletions described in G, $t \leq s$. The following constraints are necessary and sufficient for the selection of G', such that $(s - t)$ actions may be deleted from PN by the strategy set $G \cup (G ø G')$:

1. All P defined for G and G' are altered by G';
2. If strategy option G'(k) in G' is selected to alter strategy point MO(i.j) in P and G'(k) $\in$ D; then, the action deletion described by G'(k) must not be impeded by $G \cup (G \not\cap G')$, for $j \leq nsc(i,1)$.
3. No absolute conflicts exist in $G \cup (G \not\cap G')$;
4. [Deleted actions in $G \cup (G \not\cap G')$] - [impeded deletions in $G \cup (G \not\cap G')$] $\geq (s - t)$.

Proof:

If G is a set of s strategy options in D, and no absolute conflicts occur in G, then G is a set of strategy options which describes s unique action deletions. If there are t impeded action deletions in G, then (s - t) defines the number of actions which may be deleted from PN by G.

Regarding Constraint 1:

To delete (s - t) actions from PN, all P defined for G must be altered; furthermore, all P defined for strategy options employed to alter all P for G must be altered. The termination of this recursive condition is assured by the convergence of deletion conditions (Theorem 5.2) and deletion strategies (Corollary 6.1). This condition is defined by Constraint 1. The feasibility of selecting a G' which satisfies Constraint 1 is insured by the following: An exhaustive examination of deletion conditions for deletion candidates identifies all P required by $D \cup F$ and all $D \cup F$ which may be employed to alter each strategy point P(v), v=1,2,...,nP. Since for any P(v) $\in$ P, the alteration of P(v) is possible by at least one strategy option in D ∪ F, at least one G' exists which satisfies Constraint 1.

Regarding Constraint 2:

From an examination of the deletion conditions, it can be seen that strategy options are selected to effect one of two state changes in the UA: State 1|3 changed to State 0|2; or, State 2|3 changed to State 0|1 (see Section 5.2). State change 1|3 to 0|2 represents the replacement or deletion of a prime source unit reference. State change 2|3 to 0|1 represents the deletion or movement of a prime sink unit reference. If strategy option D(a) ∈ D is selected for state change 2|3 to 0|1 at UA(i,u), and D(a) is impeded by some other strategy option; then, only the number of actions deleted is changed [i.e., the maintenance of Op(i), sc(i,j), and c(i) does not affect the change made to UA(i,u), $1 \le j \le nsc(i)$]. If strategy option D(b) ∈ D is selected for state change 1|3 to 0|2 at UA(i,u), and D(b) is impeded by some other strategy option; then, the number of actions deleted is changed, and the state at UA(i,u) is unaltered. Clearly, this must be avoided, since the P(v) altered by D(b) is no longer altered after D(b) is impeded. The only case where P(v) may be altered by D(b) and then unaltered by the impedance of D(b) is where P(v) is a source unit alteration; since, from def 6.2, no nonconflicting strategy options exist which can move a sink unit definition twice, or delete a sink unit and then cause it to reappear. Constraint 2 is necessary and sufficient for the avoidance of altering and then unaltering a $P(v) \in P$. Furthermore, since all impedance relations are known for $D \cup F$, the application of Constraint 2 is feasible by the exhaustive examination of impedance relations in $G \cup (G \emptyset G')$.

Regarding Constraint 3:

The feasibility of Constraint 3 is immediate by exhaustive examination of strategy conflicts, since all absolute strategy conflicts can be determined for all strategy options $D \cup F$. Constraints 1, 2, and 3 are, therefore, necessary and sufficient to insure that G' is selected, such that $G \cup (G \emptyset G')$ alters all strategy points required by $G \cup (G \emptyset G')$, and that no absolute conflicts exist in $G \cup (G \emptyset G')$.

Regarding Constraint 4:

Since only strategy options which describe action deletions may be impeded, the deleted actions described in $G \cup (G \emptyset G')$ minus the impeded deletions in $G \cup (G \emptyset G')$ is the set of action deletions from PN. Therefore, Constraint 4 is necessary and sufficient to insure $(s - t)$ actions are deleted. Constraint 4 is feasible by an exhaustive examination of strategy conflicts, since all strategy conflicts may be determined for all strategy options in $D \cup F$.

Constraints 1, 2, and 3 insure that all required strategy points are altered. Constraints 2 and 4 insure that no absolute conflicts occur in the selection of strategy options which result in $(s - t)$ action deletion. Finally, since all variables of analysis can be determined, the implementation of the constraints is feasible.

Q.E.D.

Corollary 6.3 From Theorems 6.2 and 6.3:
If M is the optimal solution upper bound, then the existence of a strategy option set which results in M action deletions can be determined.

From Corollary 6.3 and the constraints given in Theorem 6.3, the following steps are presented as a feasible deterministic approach to the selection of an optimal strategy set. Notations established in Theorem 6.3 and Corollary 6.3 are used in these steps.

Step 1. From an examination of conflicting D elements in the SC sub-matrix, determine M, the optimal solution upper bound. Continue with Step 2.

Step 2. From an examination of conflicting D elements in the SC sub-matrix, determine a unique set G, which results in M deletions. If no such set exists, set M = M - 1 and continue with Step 2; otherwise, continue with Step 3.

Step 3. From an examination of required strategy points in the PO sub-matrix, the strategy options in the SP sub-matrix, and the strategy conflicts in the SC sub-matrix, determine if there exists a G' such that the four constraints given in Theorem 6.3 are satisfied. If such a G' exists, then $G \cup (G \oslash G')$ is an optimal strategy set selection; otherwise, continue with Step 2.

To demonstrate the approach described, an example is given next. The technique used to implement the approach is an exhaustive search of the strategy table.

6.5.2 Optimal Strategy Selection: Example

The steps described in the last section are now applied to the strategy table given in Table 6.3. The technique used, exhaustive search, is by no means intended as an example of an efficient algorithm.

An exhaustive search of the upper-left 11 by 11 sub-matrix in ST yields M = 4 as the optimal solution upper bound. Note that the strategies to delete MO(3) and MO(6) are deletion impedance conflicts, except for strategy options 2 and 7. Also, all strategies to delete MO(5) and MO(7) are deletion impedance conflicts. Thus, either MO(5) or MO(7) may be deleted, but not both; and, MO(3) and MO(6) may only be deleted if strategy options 2 and 7 are in the selected set.

Among those possible strategy sets determined in Step 2 with M = 4, the following set is discussed in terms of Step 3 examinations: (1,2,6,7). For the examination of (1,2,6,7), begin with strategy option 1, and examine the PO sub-matrix to determine that strategy points 18 and 22 require alteration. Now, examine the SP sub-matrix to determine nonconflicting strategy options for altering 18 and 22. Strategy point 18 may be altered by strategy option 2, with no conflict. Strategy point 22 may be altered by strategy 13, with no conflict. Next, examine the PO sub-matrix to determine strategy points for strategy option 2. They are 20 and 24. An examination of the SP shows that strategy point 20 may be altered by strategy option 7, with no conflict. Strategy point 24 may be altered by strategy option 15, 16, or 17. From Constraint 4 in Theorem 6.3, strategy option 15 is not possible for G = (1,2,6,7), because it is an absolute conflict with strategy option 2. Also from Constraint 4, strategy options 16 and 17 are not possible, because they are absolute strategy conflicts with strategy option 7. Therefore, with G = (1,2,6,7), no G' may be selected such that $G \cup (G \not\cap G')$ is optimal.

An examination of the SC sub-matrix shows that only sets which contain strategy options 2 and 7 may result in the deletion of M actions. An examination of the PO and SP sub-matrices shows that the alteration of strategy point 24 (required for strategy option 2) always results in a conflict with strategy option 7. Therefore, optimal deletion is impossible with strategy options 2 and 7 where M = 4. M is decremented by one, and Step 2 is initiated.

Three of many possible strategy sets G, for M = 3 are (1,2,7), (1,5,8), and (1,6,9). Clearly, G = (1,2,7) is not optimal because of the previously described conflict between strategy options 2 and 7.

For G = (1,5,8), required strategy points are 18 and 22. By selecting the strategy set G' = (12,13), an optimal strategy set is defined by $G \cup (G \emptyset G') = (1,5,8,12,13)$. The application of this optimal strategy set upon the original program node results in the optimized program node given in Figure 6.1 [note that strategy options 5 and 8 involve fan-out at I(3) and I(7)].

I(1)	GATE	{2}	{3}
I(3)	ADD	{3;4}	{7;5}
I(4)	ADD	{1;2}	{2}
I(7)	ADD	{2;4}	{9;8}
I(8)	SUB	{7;3}	{3}

Figure 6.1 Optimized Form of Program Node in Figure 3.1, With Fan-out

It is possible to restrict analysis to those strategy options which require no fan-out. For this example, the strategy set G = (1,6,9) results in an optimal strategy set that requires no fan-out strategies. The strategy set G' = (12,13,10,2,9,16) is selected to alter strategy points 18, 22, 23, 19, 20, and 24 respectively. These are all the strategy points required by G and G'. Thus, the optimal strategy set $G \cup (G \not\cap G') = (1,2,6,9,10,12,13,16)$. Note that action deletions described by strategy options 2 and 10 are impeded. The application of this optimal strategy set upon the original program node results in the program node given in Figure 6.2.

I(1)	GATE	{2}	{3}
I(3)	ADD	{3;4}	{5}
I(4)	ADD	{1;2}	{2}
I(7)	ADD	{2;4}	{8}
I(8)	SUB	{5;3}	{3}

Figure 6.2 Optimized Form of Program Node in Figure 3.1, With No Fan-out

In summary of the analysis approach, deletion strategies, strategy points, and strategy conflicts have been used to determine a strategy set which results in the optimal deletion of actions. It can be concluded that the identification of alternate action forms significantly increases the opportunities for nonessential action deletion.

As a final point, it is interesting to note that all nonessential action deletions defined by Klier and Ramamoorthy [27] may be described in terms of the ST structure. That is, those strategy options which require no strategy point alteration; are not in conflict; do not relate to an analysis of exit status [MO(N+1)]; and, which relate to parallel actions that specified the identical input memory sub-units at the source-level. In short, no strategies would exist for action deletion in the program node given in Figure 3.1.

6.6 Summary

Chapter 3 established a framework for the analysis of alternate action forms. A notation was also presented for the general description of a microprogram. Chapter 4 presented and proved an algorithm which identifies all alternate and equivalent action forms. Using these alternate action forms, Chapter 5 described a minimal set of conditions required for the deletion of a nonessential action. The application of these conditions to a program node yields a set of deletion candidates. This Chapter described the determination of all possible deletion strategies. Then, by examination of deletion strategies, strategy points, and strategy conflicts, the optimal deletion of actions was shown to be feasible.

With reference to the solution approach described in Section 3.3, Chapter 4 presented an algorithm for the identification of all forms F(m), of action(m), m=1,2,...,N. Through an analysis of F(m), Chapter 5 defined the conditions necessary to identify all candidates for deletion, D. This Chapter described the identification of strategies which may be employed in the deletion of actions in D. Then, through an analysis of conflicting strategies, it was shown that it is possible to select one form, Z(k), for each action, such that the maximum number of actions are deleted from the program node, $Z(k) \in F(m)$, $k=1,2,\ldots,N$. The next Chapter presents study conclusions and recommendations for further research.

CHAPTER VII

SUMMARY, CONCLUSIONS, AND RECOMMENDATIONS

This study posed and solved three major problems in the discipline of Microprogram Optimization. These problems are the identification of alternate and equivalent action forms; the definition of deletion strategies which include an analysis of alternate and equivalent action forms; and, the determination of a set of deletion strategies which may be employed in the optimal deletion of nonessential actions. The solution of these problems results in a significant increase in the opportunities for nonessential action deletion; i.e., actions in a program node may be altered to force other actions to assume a nonessential status.

Before the problems were addressed, a notation was presented for the general description of a microprogram. This notation was designed to include any encoding structure for a micro-instruction and time validity constraints for a micro-operation. The results of this study may be applied directly to any microprogram expressed in the proposed notation. Also, many software programming languages may be mapped directly onto the notation for a highly encoded monophase micro-instruction. Therefore, the results of this study, with qualifications, may be shared by the discipline of General Compiler Optimization.

In designing the framework for the study, it was found that the set of redundant actions is covered by the set of parallel actions. This serves to generalize the approach to nonessential action deletion by restricting examination to actions which reference the same output memory sub-unit and to actions which are equivalent.

To aid in understanding the combinatoric nature of the problems addressed, a graph-theoretic description of actions in a program node was presented. However, for the identification of alternate and equivalent action forms, a graph-theoretic approach was shown to involve needless recursive analysis of sub-graphs. Alternate and equivalent action forms may be identified by an iterative deterministic algorithm.

The first major problem was solved by the proposal and proof of an algorithm which identifies all alternate and equivalent action forms. Equivalent actions are identified by the application of a hashing function. The identification of equivalent actions allows all equivalently defined memory sub-units to be identified. The qualification of these units by their index of definition established a means of identifying all possible inputs and outputs for an action, including those action forms which may only exist after other actions have been changed or deleted.

With respect to order of computation, the algorithm requires a maximum of two examinations of each action in the program node. The primary limitation of the algorithm is that run-time statistics are not used to identify alternate action forms. It can be seen, however, that the inclusion of run-time statistics would only serve to increase the number of equivalent actions identified; the theoretical deletion advantage of alternate action identification would be unaltered.

The second major problem was solved by the definition of a minimal set of conditions which must be satisfied in order for an action to be deleted. These conditions are applied to the actions in a program node to determine those actions which may participate as candidates for deletion. By successive examination of these deletion conditions with respect to each deletion candidate, it was shown that all possible deletion strategies may be identified. Furthermore, all locations in the program node are identified, where these strategies may be applied.

A significant problem encountered in the definition of deletion conditions was that action deletions may have an order-isomorphic relationship to each other. Because these cases are identifiable in the application of deletion conditions, it was possible to formulate a means of avoiding order-isomorphisms. Thus, the convergence of deletion conditions and deletion strategies is assured.

A basis for the solution of the third major problem was established by the definition of strategy conflicts. Strategy conflicts were used to prove the existence and identification of an optimal solution upper bound; i.e., the maximum number of actions which may be deleted from the program node. Strategy conflicts were also used to prove that the existence of an optimal strategy set may be determined. Finally, a deterministic optimal strategy selection approach was described.

The major limitations upon the analysis of deletion candidates are that branch-points are not included, and program region analysis is not performed. The resumption of this study without these limitations provides a possibility for future research. Global optimization procedures which have been applied to software languages may also be applicable to microprogramming languages. For example, it may be possible to identify program node equivalencies; thereby, deleting an entire node. In a broad perspective, the results of this study would be based upon an analysis of regional properties and units which are busy on entry into each program node. The branch-point limitation presents a special problem because of the varied schemes used to select the next micro-instruction.

The implementation of the optimal strategy selection approach provides numerous opportunities for future research. From the standpoint of feasibility, the approach described is acceptable; however, there may exist better approaches, from the standpoint of implementation. For example, the strategy selection problem may be examined for conformity to a graph coloration problem (possibly convert strategy conflicts into preassignment constraints). It may also be possible to assign weights to analysis variables and approach the problem in terms of linear programming or game theory. Finally, the problem may conform to a scheduling problem where origins are units and destinations are actions.

The results of this study clearly establish a new criterion for the evaluation of program fitness: The relationship of equivalent actions and equivalently defined memory sub-units to program efficiency. It is believed, although not proved, that a relationship exists between unnecessary actions and the number of actions over which equivalently defined memory sub-units exist.

As a final recommendation, this study may have applications in the general area of hardware evaluation. Simply stated, the action deletion advantage gained by fan-out may be compared to the action deletions where fan-out is not possible. The result of this comparison may provide a valuable tool for the evaluation of advantages gained from parallel hardware units.

BIBLIOGRAPHY

1 Abrams, P. S. "An APL machine." Stanford Linear Accelerator Center, Stanford University, Stanford, Calif., Rep. SLAC-114.

2 Allen, F. E. "Program Optimization." Annual Review in Automatic Programming (1969), Vol. 5, pp. 239-307.

3 Amdahl, L. D. and G. H. Amdahl. "Fourth-Generation Hardware." Datamation (January, 1967), Vol. 13, No. 1, pp. 25-26.

4 Barlow, J. P. (Chairman: panel discussion). "Firmware Techniques." SIGmicro NEWSLETTER, ACM Special Interest Group on Microprogramming (October, 1971), Vol. 2, No. 3, pp. 22-27.

5 Bartee, Thomas C., Lebow, Irwin L. and Irving S. Reed. Theory and Design of Digital Machines. New York: McGraw-Hill Book Company, Inc., 1962.

6 Bartlett, J. P. "Processing Memories." IEEE Computer Group Conference (June, 1970), pp.229-307.

7 Berge, C. The Theory of Graphs. London: Methuen and Co., Ltd, 1964.

8 Berztiss, A. T. Data Structures Theory and Practice. New York: Academic Press, Inc., 1971.

9 Cheatham, T. E., Jr., Fisher, A. and P. Jorrand. "On the Basis for ELF - An Extensible Language Facility." AFIPS conf. proc. FJCC (1968), Vol. 33, pp. 937-948.

10 Chu, Yaohan. "An ALGOL-like Computer Design Language." Comm. of ACM (October, 1965), Vol. 8, No. 10, pp. 607-615.

11 Cook, Robert W. and Michael J. Flynn. "System Design of a Dynamic Microprocessor." IEEE Transactions on Computers (March, 1970), Vol. C-19, No. 3, pp. 213-222.

12 Dahl, Ole-Johan and Kristen Nygaard. "SIMULA - An ALGOL - Based Simulation Language." Comm. of ACM (September, 1966), Vol. 9, No. 9, pp. 671-678.

13 Davis, R. L. and S. Zucker. "Structure of a Multiprocessor Using Microprogrammable Building Blocks." SIGmicro NEWSLETTER, ACM Special Interest Group on Microprogramming (October, 1971), Vol. 2, No. 3, pp. 28-42.

14 Dietmeyer, D. L. Logic Design of Digital Systems. Boston: Allyn and Bacon, Inc., 1971.

15 Gardner, Peter L. "Functional Memory and Its Microprogramming Implications." IEEE Transactions on Computers (July, 1971), Vol. C-20, No. 7, pp. 764-775.

16 Gear, C. W. "High Speed Compilation of Efficient Object Code." Comm. of ACM (August, 1965), Vol. 8, No. 8, pp. 483-488.

17 Gluck, Simon E. "Impact of Scratchpads in Design: Multifunctional Scratchpad Memories in the B8500." AFIPS conf. proc. FJCC (1965), Vol. 27, pp. 661-666.

18 Grasselli, A. "The Design of Program-Modifiable Micro-Programmed Control Units." IRE Transactions on Computers (June, 1962), Vol. 11, pp. 336-339.

19 Green, Julien. "Microprogramming, Emulators, and Programming Languages." Comm. of ACM (March, 1966), Vol. 9, No. 3, pp. 230-232.

20 Gries, David. Compiler Construction for Digital Computers. Toronto: John Wiley & Sons, Ltd., 1971.

21 Hassitt, A. "Microprogramming and High Level Languages." Proc. of IEEE International Computer Society Conference (September, 1971), pp. 91-92.

22 Hawryszkiewycz, Igor T. "Microprogrammed Control in Problem Oriented Languages." IEEE Transactions on Computers (October, 1967), Vol. EC-16, No. 5, pp. 652-658.

23 Hendricks, M. C. "Optimization of a 16-bit Microprogrammed Digital Processor." Ph.D Dissertation, University of Denver, 1970.

24 Husson, Samir S. Microprogramming: Principles and Practices. Englewood Cliffs: Prentice-Hall, Inc., 1970.

25 Jakolat, Fred A. "Advantages of Large Micro-Program Control Words." SIGmicro NEWSLETTER, ACM Special Interest Group on Microprogramming (October, 1971), Vol. 2, No. 3, pp. 15-17.

26 Kampe, Thomas W. "The Design of a General-Purpose Microprogram-Controlled Computer with Elementary Structure." IRE Transactions on Computers (June, 1960), Vol. EC-9, No. 2, pp. 208-213.

27 Klier, R. L. and C. V. Ramamoorthy. "Optimization Strategies for Microprograms." IEEE Transactions on Computers (July, 1971), Vol. C-20, No. 7, pp. 783-795.

28 Knuth, E. E. The Art of Computer Programming, Volume 1 / Fundamental Algorithms. Don Mills, Ontario: Addison-Wesley Publishing Company, Ltd., 1968.

29 Kurpanek, Horst G. (Chairman: panel discussion). "Word Length in Microprogrammed Processors - Criteria and Minimization Techniques." SIGmicro NEWSLETTER, ACM Special Interest Group on Microprogramming (October, 1971), Vol. 2, No. 3, pp. 8-11.

30 Lawson, Harold W., Jr. "Programming-Language-Oriented Instruction Streams." IEEE Transactions on Computers (May, 1968), Vol. C-17, No. 5, pp. 476-485.

31 Lawson, Harold W. and Burton K. Smith. "Functional Characteristics of a Multilingual Processor." IEEE Transactions on Computers (July, 1971), Vol. C-20, No. 7, pp. 732-743.

32 Lazarev, V. G. "Matrix Method of Minimization of Microprogram Systems." Engineering Cybernetics (1965), No. 2, pp. 32-38.

33 Lesser, Victor R. "An Introduction to the Direct Emulation of Control Structures by a Parallel Microcomputer." IEEE Transactions on Computers (July, 1971), Vol. C-20, No. 7, pp. 751-764.

34 Llewllyn, R. W. Linear Programming. New York: Holt, Rinehart, and Winston, 1963.

35 Lowry, Edward S. and C. W. Medlock. "Object Code Optimization." Comm. of ACM (January, 1969), Vol. 12, No. 1, pp. 13-22.

36 Magleby, Kay B. "System Organization Considerations for Microprogrammed Processors." SIGmicro NEWSLETTER, ACM Special Interest Group on Microprogramming (October, 1971), Vol. 2, No. 3, pp. 11-15.

37 McKeever, B. T. "The Associative Memory Structure." AFIPS conf. proc. FJCC (1965), Vol. 27, pp. 371-387.

38 Melborne, A. J. and J. M. Pugmire. "A Small Computer for the Direct Processing of FORTRAN Statements." Comput. Journal, (April, 1965), Vol. 8, pp. 24-27.

39 Neufeld, G. A. "Analysis of Class-Timetable Problems." Ph.D. Dissertation, University of Alberta, 1972.

40 Nichols, A. J. "A Microprogramming Framework for Experimental Machine Design." SIGmicro NEWSLETTER, ACM Special Interest Group on Microprogramming (July, 1971), Vol. 2, No. 2, pp. 17-27.

41 Nievergelt, J. "On the Automatic Simplification of Computer Programs." Comm. of ACM (June, 1965), Vol. 8, No. 6, pp. 166-170.

42 Opler, Ascher. "New Directions in Software 1960-1966." Proceedings of the IEEE (December, 1966), Vol. 54, No. 12, pp. 1757-1763.

43 Opler, Ascher. "Fourth-Generation Software." Datamation (January, 1967), Vol. 13, No. 1, pp. 22-24.

44 Rakoczi, L. L. (Vice President, Systems Development, Standard Computer Corporation), Personal Interview, August 25, 1971, Santa Ana, California.

45 Ramamoorthy, C. V. and M. J. Gonzalez. "A Survey of Techniques for Recognizing Parallel Processing Streams in Computer Programs." AFIPS conf. proc. FJCC (1969), Vol. 35, pp. 1-15.

46 Ramamoorthy, C. V. and R. L. Klier. "A Survey of Techniques for Optimizing Microprograms." Presented at ACM 3rd Annual Workshop on Microprogramming, Buffalo, N. Y., October, 1970.

47 Redfield, Stephen R. "A Study in Microprogrammed Processors: A Medium Sized Microprogrammed Processor." IEEE Transactions on Computers (July, 1971), Vol. C-20, No. 7, pp. 743-750.

46 Rosin, R. F. "Contemporary Concepts of Microprogramming and Emulation." Computing Surveys (December, 1989), Vol. 1, No. 4, pp. 197-212.

49 Schlaeppi, H. P. "A Formal Language for Describing Machine Logic, Timing, and Sequencing (LOTIS)," IBM Research - Technical Report RX 125 (December 24, 1963).

50 Tsichritzis, D. "The Equivalence Problem of Simple Programs." Journal of the ACM (October, 1970), Vol. 17, No. 4, pp. 729-738.

51 Tucker, Allen B. and Michael J. Flynn. "Dynamic Microprogramming: Processor Organization and Programming." Comm. of ACM (April, 1971), Vol. 14, No. 4, pp. 240-250.

52 Tumasulo, R. M. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Journal, (January, 1967), pp. 25-33.

53 Wilkes, M. V. "The Best Way to Design an Automatic Calculation Machine," Manchester University Computer Inaugural Conference Proceeding (1951), pp. 16-31.

54 Wilkes, M. V. "Microprogramming: The Hardware Software Interface." Proc. of IEEE International Computer Society Conference (September, 1971), pp. 93-94.

55 Young, Steven. "A Microprogram Simulator." SIGmicro NEWSLETTER, ACM Special Interest Group on Microprogramming (October, 1971), Vol. 2, No. 3, pp. 43-57.