THE UNIVERSITY OF MICHIGAN

SYSTEMS ENGINEERING LABORATORY

Department of Electrical and Computer Engineering
College of Engineering

SEL Technical Report No. 59

# OPTIMAL SELECTION OF FUNCTIONAL HARDWARE UNITS FOR MICROPROGRAM CONTROLLED CENTRAL PROCESSING UNITS

by

David Leary Hinshaw

under the direction of
Professor Keki B. Irani

April 1972

Under contract with
Rome Air Development Center
Research and Technology Division
Griffiss Air Force Base, New York
Contract No. F30602-69-C-0214

## ACKNOWLEDGMENTS

The research presented in this dissertation has been made possible through the efforts of many people. Professors K. B. Irani, B. W. Arden, B. A. Galler, A. W. Naylor, and F. H. Westervelt as members of my doctoral committee provided both guidance and encouragement. I would particularly like to thank Professor K. B. Irani who acted as chairman of my doctoral committee. Special thanks must also go to Dr. M. P. Ristenbatt who provided needed support and encouragement during my first year at the University.

I wish to thank Ms. Kass Barker who cheerfully typed several final versions of the dissertation, Ms. Ann Fulmer who typed Chapter IV, and Ms. Joyce Doneth who did the final corrections.

Most importantly I wish to thank Carole. Her encouragement, patience, and understanding made this work possible.

This work was supported by the Systems Engineering Laboratory of the University of Michigan through Rome Air Development Center Contract F30602-69-C-0214.

David L. Hinshaw

Ann Arbor, Michigan
April 1972

To

CAROLE

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF TABLES

LIST OF TABLES (Continued)

# LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

# LIST OF SYMBOLS

This list contains in roughly chronological order the symbols which are used with some frequency in this dissertation.

| Symbol | Reference | Significance |
|---|---|---|
| $P$ | 3.2 | List of program type. |
| $H$ | 3.2 | The set of hardware options. |
| $B^j$ | 3.2 | The implementation matrix specifies which hardware options are required by the implementation methods for the jth program type. |
| $R^k$ | 3.2 | Some subset of H. |
| $C^i$ | 3.2 | Cost vector for ith hardware option. |
| $U^i_j$ | 3.2 | The inclusion vector specifies which options must be present if jth element of $C^i$ is used for cost of the ith hardware option. |
| $X^i_j$ | 3.2 | The exclusion vector specifies which options must not be present if jth element of $C^i$ is used for cost of the ith hardware option. |
| $\rho(R^k, U^i_j, X^i_j)$ | 3.2 | The $\rho$ function has a value of 1 if $R^k$ satisfies the requirements specified by $U^i_j$ and $X^i_j$. |

| Symbol | Reference | Significance |
|---|---|---|
| O | 3.2 | The occurrence vector gives an estimate of the number of occurrences of each program type in a typical program. |
| F | 3.2 | The frequency vector gives an estimate of the frequency of execution of each occurrence of each program type in a typical program. |
| $V^j$ | 3.2 | The execution time vector is the microcycles required to perform the jth program type function for each implementation method. |
| $M^j$ | 3.2 | Microbit vector for jth hardware option. |
| w | 3.2 | The number of wordforms used by the base hardware. |
| $W^j$ | 3.2 | The word form vector specifies to which of the word forms the microbits required by the jth hardware option are to be added. |
| $c_b$ | 3.2 | Cost of the base hardware. |
| $c_s$ | 3.2 | Cost of a bit of control store. |
| $m_b$ | 3.2 | Base microbits. |
| k | 3.2 | Maximum number of bits per field (field encoding limit). |

| Symbol | Reference | Significance |
|---|---|---|
| E | 3.2 | The subroutine selection vector indicates which program types will be implemented as in-line code and which will be implemented as subroutines. |
| $s_B$ | 3.2 | Number of microinstructions to initiate a branch. |
| $s_R$ | 3.2 | Number of microinstructions required to initiate a return from a subroutine. |
| $S^j$ | 3.2 | The storage vector lists the number of microinstructions required for each implementation of the jth program type. |
| $N^i$ | 3.2 | Selection vector lists the current choice of implementation method for each program type. |
| CH | 3.2 | Cost of hardware options and base hardware. |
| $M(R^\ell)$ | 3.2 | Width of microinstruction word. |
| CNS | 3.2 | Cost of control store for nonsubroutine implementations. |
| CSS | 3.2 | Cost of control store for subroutine mplementations. |
| TNS | 3.2 | Execution time nonsubroutine implementations. |
| TS | 3.2 | Execution time subroutine implementation. |

| Symbol | Reference | Significance |
|---|---|---|
| $\bar{T}(N^{\ell})$ | 3.2 | Average execution time for an implementation set. |
| PT | 3.2 | Performance $1/\bar{T}$ |
| $P_r$ | 4.3 | Required performance. |
| $C_{MT}$ | 4.3 | Cost of hardware options and minimum control storage. |
| X | 4.3 | Hardware option set. |
| $C_T$ | 4.3 | Cost of hardware options and storage for fastest implementations. |
| $C_m(x)$ | 4.3 | Minimum cost of hardware point x. |
| n(x) | 4.3 | Numerical ordering of x. |
| h(x,x') | 4.3 | Hardware options of hardware option points x'', $n(x) \leq n(x'') < n(x')$. |
| NP(x) | 4.3 | Computes x' the next hardware option point after x in numerical which could have a cost less than x |
| $\Delta t^{j}_{\ell k}$ | 4.3 | Increase in execution time if $\ell$th implementation method for jth program type is replaced by the kth implementation method for the jth program type. |
| $\Delta s^{j}_{\ell k}$ | 4.3 | Reduction in control storage if the $\ell$th implementation method for the jth program type is replaced by the kth implementation method for the jth program type. |

| Symbol | Reference | Significance |
|---|---|---|
| dT | 4.3 | Total increase in execution time. |
| dS | 4.3 | Total reduction in storage. |
| $a_{ji}$ | 4.3 | The ith implementation of the jth program type. |
| $D_i$ | 4.3 | The total number of implementations of the jth program type. |
| $\alpha$ | 4.3 | Set of all program type implementations. |
| $\beta$ | 4.3 | $\beta \subset \alpha$ |
| $\beta^j$ | 4.3 | $\beta^j = \{a_{ji} \mid a_{ji} \epsilon \beta\}$ |
| I set | 4.3 | I set contains one and only one element from each $\beta^j$ set. |
| $I^1$ set | 4.3 | Set of fastest implementations for a hardware point. |
| $T_{max}$ | 4.3 | Maximum acceptable increase in execution time. |
| $S_{req}$ | 4.3 | Required savings in control storage. |
| $I_N$ | 4.3 | I set after Nth iteration of slope selection algorithm. |
| $S(I_N)$ | 4.3 | Number of words of control storage required by implementation set $I_N$. |
| $T(I_N)$ | 4.3 | Number of microcycles to execution implementation set $I_N$ a number of times defined by F and O. |

| Symbol | Reference | Significance |
|---|---|---|
| $\Delta T(I_N)$ | 4.3 | $\Delta T(I_N) = T(I_N) - T(I^1)$ |
| $\Delta S(I_N)$ | 4.3 | $\Delta S(I_N) = S(I^1) - S(I_N)$ |

ABSTRACT

Optimal Selection of Functional Components for
Microprogrammable Central Processing Units
by
David Leary Hinshaw

Chairman: Keki B. Irani

The introduction and the wide acceptance of both microprogram control and medium to large scale integrated circuitry have drastically altered the manner in which central processing units are implemented. The use of microprogram control and the tendency to acquire hardware in functional units allows a microlevel hardware software tradeoff to occur during the design of the central processing unit. The combined effect of microprogram control and the availability of powerful functional units has been to alter the CPU design environment. The overall goal of this research is to develop a design philosophy which is applicable to the new CPU design environment. The work proceeds in four steps: 1) the development of the general design method; 2) the initial development of the Program Type Language (PTL); 3) the development of the CPU model; and 4) the development of an efficient optimization procedure.

The Program Type Language is an initial version of a machine independent intermediate language. The language is designed to meet two principle objectives. First, the language allows valid statistics to

be gathered on the functional requirements of the computer user. Each program type is selected to provide a basic function which is a major building block of a highly used computer area. Second, the language is machine independent and it meets the requirements of an UNCOL language.

The CPU is modeled as a base hardware set plus a set of hardware options. The base hardware set defines a general architecture and the hardware options allow major design variations about that architecture. The hardware options consist mainly of functional units and the use of the hardware option concept in the model is desirable in view of the rapid acceptance of large and medium scale integration. The cost of a hardware option is dependent upon the choice of the other hardware options as is the number of microbits required to control each hardware option. The specification of microinstruction word size is very flexible, allowing both field encoding and word encoding to be modeled.

The design method is user directed. The user writes or translates his programs into PTL. Since, PTL is constructed to allow valid statistics to be gathered on user requirements; PTL provides a mechanism for determining user requirements. The design method makes optimum use of the human design. The connection between the program types and the CPU model is the implementation methods. There is

one or more implementation method for each program type. Each implementation method specifies a CPU configuration. The human designer specifies the implementation method and is concerned with performance - cost (hardware) tradeoffs at the program type level. The optimization program performs the performance - cost tradeoffs for the entire set of program types using the individual implementation methods supplied by the designer. The designer does not have to attempt to evaluate the combined effort of adding an optional piece of hardware on 100 or more program types. He can focus his attention on each program type one by one.

The optimization technique involves three distinct parts. First, the use of bounding techniques allows the search procedure to be applied to the hardware option space (typically H space $< 10^6$) instead of the implementation space (typically I space $> 10^{12}$). Second, a modification of the Lawler Bell technique is combined with a bounding procedure to eliminate large sections of the hardware option space. Third, a slope selection technique was developed to select the optimal implementation set for any fixed hardware option set.

# Chapter I

## INTRODUCTION

This study concerns itself with the design of central processing units. It is directed at the CPU design environment which is rapidly coming into existence. This new environment is influenced by four developments. The four developments are the increasing use of microprogram control, the decreasing cost of microcontrol storage, the increasing availability of both medium and large scale integration, and the increasing ratio of software development to hardware development costs.

The changing environment will have an effect on CPU design. The effects caused by the developments mentioned above are interrelated but can be broadly classified as follows. The effect of medium and large scale integration is to force the design of computer systems which are composed of powerful functional blocks. The effect of microprogram control and inexpensive control storage is to allow hardware software tradeoffs to be made at the control storage level. The effect of the increased ratio of software development to hardware development will be twofold. First, more powerful instruction sets will be provided. This tends to move software functions into the CPU design area. Second, different approaches will be developed to allow software to be transferred

from computer to computer.

The work presented here is directed toward the development of the tools required to optimally design computers in the new environment. The effort is extended over a broad area (languages, computer models, optimization) and culminates in a computer aided design method for the optimal selection of functional components for microprogrammable central processing units.

The work is divided into several interrelated parts. First, a general design method is proposed. The design method requires an accurate method of predicting the operational requirements to be placed upon the microprogrammed computer. The operational requirements are the functions which the computer user wishes to perform.

Second, a new language is proposed. The language is designed to allow accurate measurement of the operational requirements of programs written in the new language. Also, the language is designed to be both easily used and efficiently converted into microcode.

Third, a cost performance model of a microprogrammed central processing unit is developed. The model views the central processing unit as a base hardware unit plus a set of optional hardware units which are used to implement a set of program types. The program types are the operations of the new language. Each program

type may have several implementations with each implementation requiring a different set of hardware options.

Fourth, an optimization procedure is developed. The procedure allows the very large state space resulting from combinations of implementation methods and hardware options to be searched efficiently. The procedure selects the minimum cost set of hardware options and implementation methods which achieve a desired performance level.

Last, several examples are presented.

## 1.1 Related Research

The area of microprogram control and automated design are both related to the work presented in this report. There has been significant interest in both subjects and the history and advantages of each have been reported in numerous articles.

Some excellent references on the subject of microprogram control are Tucker (T2, introduction to microprogramming), (T1, emulation of IBM 360 series of computers), Flynn (F2, a key paper which stimulated interest in user microprogramming), Wilkes (W3, a historical review by the inventor of microprogramming), Rosin (R3, a good explanation of microprogramming and a strong supporter of user microprogramming), Davis (D3, a recent article with an annotated bibliography - IBM viewpoint), and Husson (H5, a book on the subject with excellent material on existing microprogrammable computers). Throughout the literature the terms vertical and horizontal microprogram control have been used to describe two broad classes of microprogram control. In addition, various authors have introduced "special terms" to describe particular concepts or physical implementations of microprogram control. Appendix C provides an introduction to microprogram control, gives a definition of vertical and horizontal microprogram control, and places the "special terms" in perspective.

The optimal selection of functional components can be viewed as a hardware (functional components) firmware (microcontrol) tradeoff. It is apparent that hardware firmware tradeoffs are made any time a microprogrammable computer is designed. However, there is no research which has been reported upon in the literature which examines the question of hardware firmware tradeoffs. The work reported upon in the literature has been mainly concerned with the optimization of performance by moving machine language routines into microcode (F2, H1, H2, R1, R3, S7, T3), the specification of the type of microprogram control which would achieve a speed cost tradeoff for a fixed architecture (G1, R2, R4), or the improvement of diagnostics via microprogram control (B1, C1, J1).

The general area of design automation is very large and expanding. The most comprehensive survey of the area has been given by Breuer in two papers (B5, B6) the most recent printed in January, 1972. It is interesting to note that Breuer predicted, as one of a number of predictions, that synthesis programs would have to be developed which would work with standard complex functions (i.e., function units).

There are no published papers that deal with the problems which are studied in this dissertation. Many of the papers are concerned with automated wire routing, card placement, card layout, etc. The papers

which deal with CPU design at a higher level are mainly concerned with the development of computer description languages (B3, D2, D5, G3, S2, S6), and the development of languages to describe the input output behavior of computers (B3, D6, F5, G2, M3, P1). None of the studies attempts to optimize the computer design in the sense of performing performance cost tradeoffs. The hardware design routines supporting the languages tend to provide a direct translation of the language descriptions.

## 1.2 CPU Design Method

The design method considered here attempts to combine the human designer and computer design aids in the most efficient fashion. The method is composed of four steps. First, a set of operations called program types is determined. The program types are the operations which the computer will be designed to implement. Second, statistics on the usage of each program type are gathered. The statistics on program type usage allow the computer to be designed with the performance calculations weighed according to program type usage. Third, the computer designer provides several different hardware combinations, each with different cost and performance figures, for each program type. Fourth, an optimization program selects and combines the multiple program type implementations provided in step three. The output from the optimization program is the minimum cost computer hardware configuration which achieves some desired average performance.

The design philosophy (Figure 1.2.1) is implemented in the following fashion. The program types are provided as part of a proposed language. The language, which is called the Program Type Language (PTL), is described in Chapter II.

A new language was proposed instead of using an existing language because it was felt that no existing language combined the features (see Chapter II) which we required. The language is designed to allow valid statistics to be taken on operations which a user wishes to perform. It is not expected that all programs will be written in the new language, just as not all programs are currently written in machine languages. However, compilers, interpreters, and other heavily used programs will probably be written in PTL, with the remaining programs compiled into PTL.

The statistics which we gathered specify the frequency with which an operation occurs in a typical program and the frequency with which each program occurrence is executed. The statistics would be gathered on programs for a particular computer installation or a particular set of users.

The statistics are used by the optimization program to allocate resources (hardware options) to maximize average operation performance.

As was stated before, the computer designer is not concerned with the efficient implementation of the Program Type Language as a whole. Instead, the designer is concerned with the implementation

of each individual operation of the language. The computer designer works with a hardware base and a set of hardware options. Such a model (the model is developed in Chapter III) seems reasonable in the light of the rapid acceptance of medium and large scale integration.

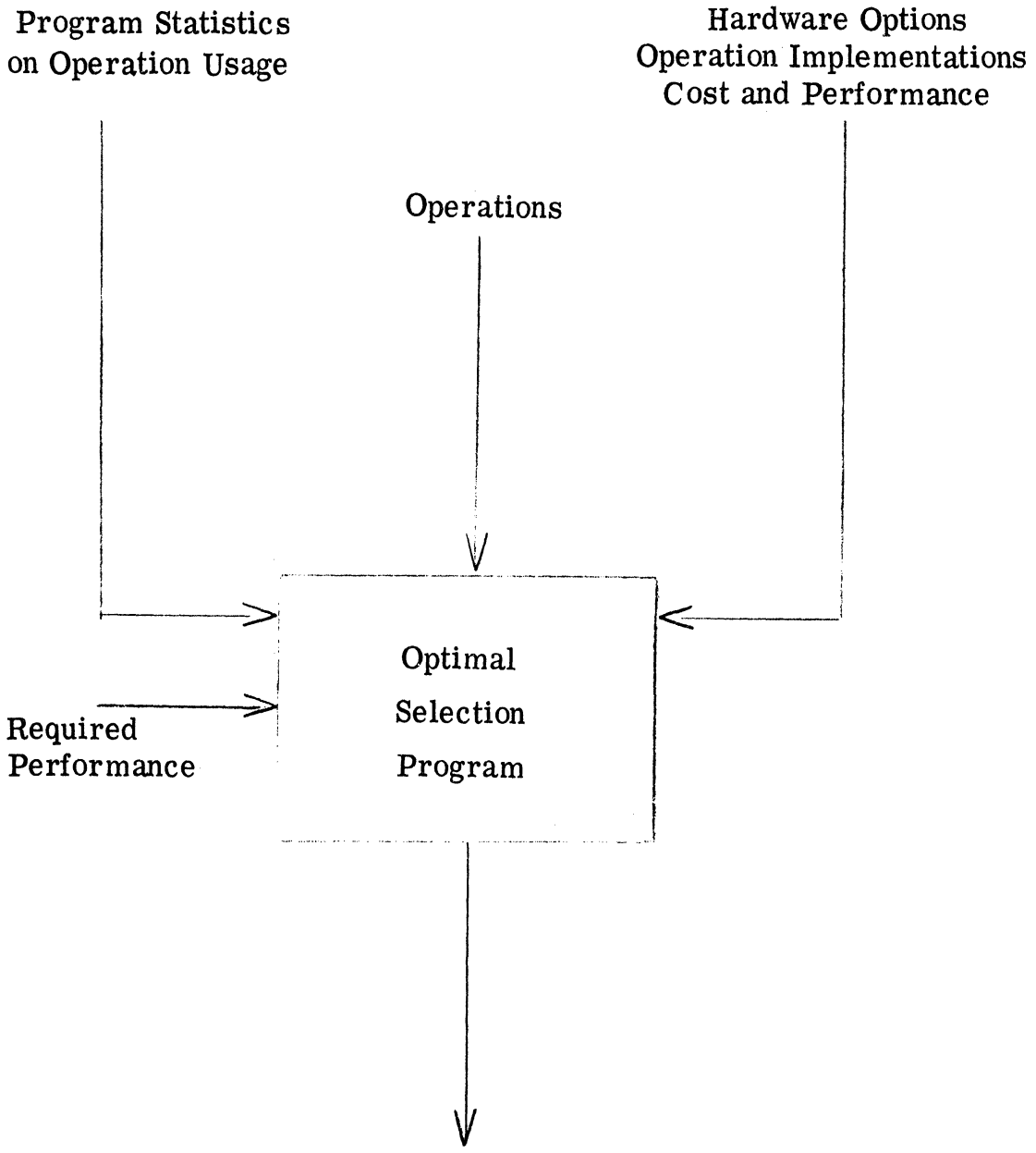
The designer specifies several different implementations for each program type operation. Each different implementation of the same operation uses different sets of hardware options and would have different cost and performance figures. Thus the designer is working at the operation level and does not have to be concerned with tradeoffs effecting the implementation of the entire language. This elimination of one level of complexity should enable the designer to work more efficiently. The implementation methods, their required hardware options, and performance figures form the second input to the optimization program.

The final input into the optimization program is a performance requirement. The performance requirement is given in terms of the average execution time of the operations of the Program Type Language. After accepting all inputs the optimization program picks the minimum cost system which will satisfy the performance requirements.

The CPU design method is a part of the overall system design

philosophy. The language is machine independent; thus, programs written in the language (in particular, compiler, interpreters, etc.) may shift from machine to machine requiring only a retranslation. The use of the implementation method concept is very useful in the translation of the programs into microcode. The implementation methods are actually microcode routines. Therefore, the operations may be replaced in-line by the implementation microcode. The assignment of variables to storage presents some problems. The estimated execution time of each program type assumes that the operations are located in high speed storage. Thus, any extra data movement from core storage to high speed storage would not be accounted for in the performance computation unless an estimate of the number of such data transfers was known a priori. Thus the optimum assignment of data to high speed registers is important. However, the extension of existing register assignment algorithms should be sufficient (see Section 2.5).

It is hoped that the design philosophy presented in this section will come into better forcus as the language chapter (II), the model chapter (III), the optimization chapter (IV), and the examples chapter (V) are read.




CPU Design Method

Figure 1.2.1

## 1.3 Two Methods of Computer Design

A description of the traditional method of computer design will be presented and compared with the computer design method presented in this dissertation. Because the computer field is changing very rapidly and because there are many different groups of individuals designing computers there are exceptions to what we describe as the traditional approach to computer design. However, most of the computer systems in existence today were designed in a manner similar to what will be called the traditional approach.

The two methods arise from different hardware environments. The traditional approach developed when instructions were implemented via hardwired circuitry. The approach we are advocating assumes instructions are implemented via microprogram control. Thus the cost constraints imposed upon the two methods are different.

The two computer design methods differ in the choice and implementation of instruction sets. In the traditional approach, the computer is designed to implement a limited set of machine or assembly language instructions. The instruction set usually includes arithmetic, logical, branching, and I/O instructions. The implementation of more complicated operations (for instance, list manipulation or square root) is usually accomplished in software at or above the assembly language level. The hardware implementation (assembly language) of

a more complicated instruction occurs only if the instruction has extraordinarily high usage. The most obvious examples of such "special" instructions are floating point instructions.

The more complicated instructions were implemented infrequently for two reasons. First, the cost of implementation was prohibitive. Second, the requirements of the user were not well known. Thus, the problem is one of measuring or in some fashion asserting what the user wishes to do. The user's real desires are often masked because he has been forced to modify his methods to conform to a limited instruction set.

In the method which we are suggesting, the first step is to try to determine what the user would truly wish to do. This decision is arrived at by studying higher level languages. Since higher level languages are designed specifically to allow a user to solve his problems in a natural fashion, the languages should represent a good measure of what people would like to do. In addition, attention will be paid to studies of compilers, sorting routines, graphic displays, etc. to determine what operations are required to implement these functions. The net result is a set of operations (program types) which are combined into an intermediate language. Each program type is a function which would be useful to a large user class. The set of program

types covers the set of operations typically offered by an assembly language and much more.

The set of program types which will be proposed in Chapter II is not expected to be the ideal set or the final set of program types. It is expected that the set of program types will evolve and grow through feedback from users. Since program types are implemented via microprogram control, the modification and addition of program types should not be as difficult as it would be if they were implemented via hardware control.

The two approaches evolve as follows. In the traditional approach, a machine language is developed and a computer is designed to efficiently implement the machine language. However, the machine language is not powerful in the sense of performing in one instruction most operations common to higher level languages or compilers, or execution systems. Therefore, each higher level language must be translated by a compiler unique to the computer's machine language into machine language.

In the method we are suggesting a powerful intermediate language is used. The language is developed with the user in mind and has many operations common to higher level languages. For each computer, one program is developed which translates programs coded in the intermediate language into microcode. Since

the computer was designed to efficiently implement the operations of the intermediate language, such programs should run efficiently. Higher level languages are translated into the intermediate language and then into microcode. The match between the higher level language and the intermediate language should be good because of the power of the intermediate language. The total number of high level language compilers should also be reduced (this is the UNCOL (S7, S8, S9) concept and is explored further in Chapter II).

With the traditional approach the assembly language is defined, and the higher level languages are defined. However, there is often a mismatch between the higher level languages and the assembly language, and thus a mismatch between the user and the computer. In our approach the intermediate language (program types) is chosen. This language contains those operations which a large set of users wish to do. Then the computer is designed to efficiently implement the intermediate language. Since the intermediate language is well matched to the user there is a good match between the user and the computer.

# Chapter II

## PROGRAM TYPE LANGUAGE

The Program Type Language (PTL) is presented in this chapter. Actually, what is presented in this chapter is a language concept. PTL is the first solid representation of the concept. It is felt that PTL will change and grow, molding the language concept to the user's needs, through actual usage.

The language concept and the resulting restrictions placed upon PTL are discussed in Section 2.1. In Sections 2.2 and 2.3 the Program Type Language is presented. In Section 2.4 the differences between PTL and other languages are discussed. In particular, the necessity for a new language instead of an existing language is discussed. In Section 2.5 the problems associated with the translation of PTL program into microcode are discussed.

## 2.1 Characteristics of PTL

Programs written in PTL will be utilized for two different purposes. First, the programs, program statistics, and statistics on the expected use of the programs will be used to select optimal hardware configurations for central processing units. Second, the programs will be compiled into microcode. The first use of PTL will allow the design of CPU's (see Section 1.2) which are optimized for certain groups of programs or are optimized for heavily used operations. The second use of PTL will allow programs to execute with greater efficiency since programs written in microcode have been shown (T3, H4) to run faster than similar problems coded in machine languages. It is expected that compilers, interpreters, and other frequently used programs will be written in the Program Type Language and then will be translated into microcode. Thus, the language is designed for users working below the level of problem or procedure oriented languages.

The Program Type Language must be designed to enhance the two major uses of the language. First, PTL must allow realistic statistics to be gathered on those functions a programmer is actually trying to implement. Second, the language should facilitate the translation of PTL programs into efficient microcode.

What other features should be embodied in the language? The language should be easily used. The language should emphasize

the basic, but often powerful operations required by the user.

Thus, the new language (PTL) should embody the following features.

(1) allow valid functional statistics to be gathered,

(2) be easily translated,

(3) be machine independent,

(4) be easily used,

(5) emphasize basic-operations.

We will discuss each requirement in more detail in the following paragraphs.

1. Statistics

The statistical information on functional usage (add, and, link list, etc.) within PTL programs is utilized to optimally select CPU hardware components. Therefore, the language must be designed to allow valid functional statistics to be taken on PTL programs. Otherwise, the selection of hardware options will not be optimal. For example, if the contents of a data quantity are to be shifted left one place and shift operations are not available in the language, the shift may be performed by a multiplication by two. However, programs written in such a language would not provide a good measure of those operations a programmer wishes to implement. Any time a

programmer implements a shift operation, the language masks his intent since the operation must be implemented in an indirect manner.

PTL was developed both by studying other programming languages, and by studying the potential applications of the language. The basic operations, required for arithmetic functions, string manipulations, bit manipulation, linked list manipulation, binary tree manipulation, stack manipulation, iteration control, branch control, and other operation types, are included in the Program Type Language. This implies that the basic operations required to write scientific programs, compilers, interpreters, graphic programs, and many other program types (we believe most other program types) are included within PTL. Therefore, programs written in the Program Type Language should provide good statistics on the basic functions required by the program.

There is a limit to the number of program types which will be provided by PTL. The problem is one of limiting the number of program types and still retaining the ability to gather realistic statistics on functional usage. To understand the problem better, consider what effect incorrect functional statistics might have on the selection of hardware components. First, suppose that shift operations are not provided as PTL operations. Therefore, if a programmer requires shift operations they will be written by him in some manner using combinations of adds, multiplies, etc. Thus, the optimization program would not select a hardware configuration which included a

hardware shift option, even if the programs required a large number of shifts, because the shifts were done via adders, multipliers, etc. This is an example of not including as a PTL operation a function which could be implemented directly and easily in hardware. Thus, such a hardware option (function) would not be chosen in the optimization process because there is no way to deduce from the programmer's expansion that the function would be utilized.

As a second example, consider the function, multiply, and let us suppose that multiply hardware is not available. Therefore, the function will have to be implemented using adders and shifters (assuming now that both operations are available). Thus, if no PTL multiply operation were available the selection of hardware would not be affected since the multiply operations would be implemented by the programmer using shifts and adds and their frequency of occurrence would be measured and utilized in the optimization. Thus, in that sense the optimization would be acceptable. However, two problems occur. First, since the programmer specifies the implementation of multiply by utilization of other PTL operations, any inefficiency introduced by the selection of an inappropriate algorithm can not be removed by the optimal hardware selection procedures and will remain. Second, there may be introduced additional overhead in coupling the individual microcoded shift and add routines. That is,

if each operation (shift and add) were implemented as microsubroutines the branching overhead would be increased since several calls to shift and add routines instead of one call to the multiply routine would be made. As pointed out in Section 2.5, the use of microsubroutines will turn out to be limited, and therefore, the inefficiencies introduced by additional microsubroutine linkages will be minimal.

With the preceding examples in mind, the selection criteria for program types will be explained. The program types attempt to span all functions which could be implemented as one microcycle hardware options. Thus such functions as add, shift, logical disjunction, complement, etc., are provided. This selection rule relates to the first example above where the effects upon hardware selection of not providing a shift operation were explored. However, some common sense must be used. One could specify an endless number of functions which perform Boolean operations on bits i, j, and k, and store the result in bit n of some word or words. All such functions could be done in one microcycle, but most would have a low utility and would clutter the language.

Thus, a second criterion of utility is coupled with the first criterion. If an operation was expected to have a high usage (i.e. high utility) it was included in the language even if it was felt that

the operation could not be provided as a hardware option. If an operation could be provided as a hardware option but was felt to have zero utility or expected use, it was left out of the language.

One final point should be made. The user is allowed to add operations to the language. The addition of operations can be accomplished in at least two ways. The language could be made extendable by the addition of a definition facility. The addition of a definition facility should be feasible if the statement formats are limited to those currently available in PTL. The definition facility would allow the definition of a new operation NAME and a list of operands. Then the operation would be defined in terms of existing PTL operations using the list of operands specified in the format definition statement. Such a feature would allow a user to make PTL conform to his particular application. The definition facility would not affect the statistics and/or the hardware optimization since the frequency of occurrence of a newly defined operation could be reflected back to its component PTL operations. It is true, as was discussed in the previous paragraphs, that any inefficiency introduced by the selection of an inappropriate algorithm can not be removed by the optimal hardware selection procedure.

The language can also be extended by providing a new implementation facility. The new implementation facility uses both the

programmer and the program type implementation designer. With the new implementation facility the programmer specifies a new operation and the implementation designer provides a microcode implementation for the particular machine the programmer has available. The addition of a new implementation facility should be feasible if the statement formats are limited to those currently available in PTL. The method requires that the PTL translator be expandable. The translator must be able to accept a new operation and its implementation microcode without rewriting the translator. The advantage of using a new implementation facility would be the guarantee of an efficient implementation for the new operation . Unfortunately, a PTL program with a new operation could be executed only on computers for which the new program type had been implemented. Thus, the ability to move PTL programs from one computer to another computer would be reduced.

The definition facility and the new implementation facility are applicable to different problems. The definition facility is useful to define operations which are efficiently implemented using combinations of existing PTL operations. The operations defined by the definition facility allow a programmer to write his program in a simpler form. The new implementation facility is useful when an operation is required which can not be efficiently implemented (in terms of microcycles to execute) using combinations of existing PTL operations and should be implemented directly in microcode.

## 2. Translation

The language should aid, not obstruct, the translation of PTL programs into efficient microcode. PTL does not have any features except named data quantities and condition sets which require special effort to translate; however, both problems are solvable. The translation problem is discussed in Section 2.3. PTL is an intermediate language, and the programmmer is required to formulate his problems in a rather strict fashion. For example, the language does not provide automatic conversion of data types. Thus, the programmer is forced forced into an awareness of any conversions which occur in his program. Automatic conversion of data types is a convenient feature and should be provided in higher level languages. However, PTL is not intended for people who write in higher level languages. Therefore, since data conversions are time consuming, they are not done automatically but must be specified by the programmer in the hope that not as many will be used. The language allows the use of temporary storage locations which do not have to be declared. Therefore, the temporary storage variables are easily used and provide an effective and natural method of informing the translator of data items which are required only temporarily.

## 3. Machine Independence

The language should be machine independent for two reasons. First, if the language is machine independent only one new compiler (a PTL compiler) will have to be written for each new machine. Second, since statistics on program utilization and statistics on PTL operation usage are used to select hardware, the language should be free of any machine dependence.

The concept of reducing software development costs through the use of a machine independent intermediate language is not new. Probably the best documented arguments for the use of such a language are those presented to justify the UNCOL concept. The Universal Computer Oriented Language was reported upon [S10, S11] in 1958 by the Share Ad-Hoc Committee on Universal Languages. Their concept consists of three levels of languages. The highest level are the POL's (Problem Oriented Languages), the second level is the UNCOL language, and the lowest level is the machine language level (ML's). The idea is to provide only one translator for each higher level language. Each POL translator would be written in UNCOL [or PTL in our case] and produce its output in UNCOL [PTL]. In addition there would be required only one translator for each computer. That translator would compile the UNCOL language [PTL] into machine language [ML].

Thus, a compiler for a POL is written in UNCOL [ PTL] and produces code in UNCOL (PTL). The compiler can be run on any machine which has an UNCOL [ PTL] to ML translator. If a new computer is developed, existing software can be used by developing a translator [ UNCOL [ PTL] to the new machine's ML] and recompiling existing software into the new ML.

The potential savings in software development costs are significant. For example, if there are N higher level languages (POL's) and M computers with M different machine languages ( ML's), then using the machine independent intermediate language (PTL) i.e., the UNCOL concept, only N + M translators must be developed versus N×M translators using the traditional higher level language to machine language concept.

There is a cloud of doubt hanging over the previous discussion. If the UNCOL concept is a valid idea, why aren't people using the UNCOL language? The answer is simple. The UNCOL language was never fully developed. There were at least two attempts [ S9, C2] to start an UNCOL language but both were not significant efforts and produced languages considerably below the level of sophistication required for an UNCOL type language. In discussions with T. J. Steel, one of the authors of the Share report on UNCOL and the developer of one of the preliminary versions of UNCOL, the

question of the demise of the UNCOL language was discussed. Two major reasons for the failure of UNCOL were given. First, at about the time the UNCOL language was under consideration people became very optimistic about new techniques for compiler generation. Many people felt compilers would be produced very cheaply and very rapidly. Thus, part of the economic motivation for UNCOL was lost as was funding for its development. Second, at the same time there was an interest in higher level languages, in particular, the idea of the development of one higher level language. Thus, if one higher level language was going to be developed there would be no reason to develop one universal intermediate language.

Is there a need for a PTL-like language? Jean Sammet in her book on programming languages [ S1] states that the motivation for UNCOL still exists. She calls UNCOL a significant but unimplemented concept. The economic motivation for such a language exists. Software is an ever increasing portion of the cost of computer system development. Higher level languages and problem oriented languages are appearing in ever increasing numbers. The need for a PTL-like language, that is, a machine independent intermediate language, does exist.

If the language has machine dependent features, any hardware selection procedure would be biased toward the machine dependent features of the language. For example, if the language contained

integer operations but did not contain floating point operations, the hardware selection procedures would never select floating point hardware. In this sense, the language would be machine dependent since it would be efficiently implemented only on machines with integer hardware units. These features are operation features and have been considered in the section on statistics.

Thus, we are considering in this section only those features which allow PTL programs to be easily changed from one machine to another.

The following features have been incorporated into PTL for machine independence. The language uses named data quantities and there are no references to machine dependent features such as registers, local storage, or even main storage. The microtranslator will be responsible for the assignment of named data quantities to machine storage areas for a particular machine. The I/O commands are simple and cause the transmission of either data or I/O signals. This simplicity should allow reasonable I/O device flexibility and still provide independence. There is no standard precision assigned to floating or integer variables. The required maximum precision for each variable or sets of variables is assigned by the programmer. Thus, the language does not assume some standard word length for data.

4. Easily Used

The language should be easy to use. What is a convenient, easy to use language means different things to different people and depends heavily upon which problems are being solved. Fortran is easy to use if mathematical expressions of the form $(x^2 \cdot y)/2$ are being written. Fortran is more difficult to use if expressions of the form

$$\sum_{i=1}^{x(j)} s_{i_j} \cdot e^{x_i}$$

are being written. Fortran is very difficult to use if one wishes to interchange the 3rd and 4th bits of an integer format word in the computer memory.

PTL is not easily used in the sense of writing FORTRAN type arithmetic statements. An expression such as $(x^2 \cdot y)/y^2$ would require several PTL instructions to implement. However, the DO operations of FORTRAN are provided in PTL. In addition, there are operations which manipulate linked list, binary trees, and character strings. Therefore, the language could be said to be useful in the following sense. A large set of basic operations is provided where each subset provides a powerful set of instructions for a particular problem area.

## 5. Basic Operations

The language should employ the basic but often powerful operations required by the user. This is essentially a restatement of 1 (statistics), 2 ( clearly translated), and 4 (easily used). What is really being implied is a limit on 1, 2, and 4. We do not want to supply every function possible. This would result in too many different operation types which would complicate both the hardware selection process and the optimization process. Thus, the suggested set of operations tries to cover each area with a trade off between user convenience and a limited number of operations. The string manipulation operations of PTL serve to illustrate the point we have been trying to make.

The string manipulation instructions have evolved from study of the SNOBOL language. The instructions are the basic instructions required for string manipulation. They allow a search for a pattern match, deletion, insertion, replacement, and concatenation of strings. The instruction set is not as full as SNOBOL and hence not as easily used. Such SNOBOL operations as alternation, and conditional value functions are not present in PTL. However, the functions performed by alternation and conditional value instructions can be accomplished by using two or three PTL instructions. Thus, the PTL language is useful for string manipulation since the string manipulation operations

performed in SNOBOL can be performed using PTL with a moderate increase in effort. The stack operation, bit operation, iteration control, and the other operations of PTL were generated in a similar fashion.

## 2.2 Data Representation

The Program Type Language (PTL) is a machine independent language. There are no references to registers, memory locations, etc. Data in the PTL language is represented as named quantities. Named data quantities may have as a name any combination of 10 or less alphanumeric characters. The first character of a name must be an alphabetic character. Each named quantity (data variable) has one or more attributes. The attributes are assigned to a named data quantity by a compile statement (2.3.0). Compile statements will be discussed in section 2.3. For the present we will discuss the attributes and the use of named quantities without considering how the named quantities are generated.

The set of attributes divides naturally into two classes named type I and type II attributes. Type I attributes are concerned with the internal representation of data. Type II attributes are concerned with the internal organization of data collections.

The data representation attributes (type I) are listed in

Table 2.2.1. Each named data quantity may be assigned at most one type I attribute. A named data quantity will accept data of only one internal representation.

The length or precision (whichever is appropriate) of a named quantity must be specified concurrent with a type I attribute. The maximum length or maximum precision should be specified. Precision is specified by the maximum number of decimal digits. Length is specified by the maximum number of bits (binary string) or characters (character strings) which will be stored in the named data quantity.

Care should be exercised in the specification of length or precision for two major reasons. First, if the maximum length of character strings, bit strings, or decimal data, is overestimated the storage requirements will be exaggerated. Second, the estimates of precision will be used to assign either single or double precision operations upon floating and integer data, and therefore, affect not only the storage requirements, but also the execution speed.

The previous statement presupposes that PTL programs are run on a conventional machine which has single precision and double precision floating hardware. The translator would use the precision assigned to each variable to select either single precision or double precision operations. If the PTL program were translated to a machine which had no floating point hardware, then the translator would select microroutines with varying precision.

One of the problems associated with the design of a machine independent language is the specification of the length or precision of variables. The length (precision) assignment for each named data quantity in a PTL program is made by the programmer. The programmer assigns a maximum length (precision), but the PTL translator is allowed to select a length (precision) which is greater than that requested. If a PTL program which specified a precision of 35 bits were to be translated onto a machine which had operations with a single precision of 32 bits and operations with a double precision of 64 bits, the named data quantities would be assigned by the translator to be double precision. If the program had a named data quantity with a precision of 65 bits, special routines would be required since the machine could not directly perform operations with a precision of 65 bits.

The method which the PTL translators will use to assign an internal length or precision violates machine independence in the strict sense. A PTL program which is run on different machines would produce different results in the least significant digits. However, if the programmer has specified the maximum required lengths (precision) correctly, the answers should be correct to the desired significances on both machines.

The data location (DL) attribute is used to identify a named data quantity in which the address of other named data quantities may be stored. This allows the use of indirect addressing in accessing data. If a named data quantity with the DL attribute is used in an operation statement, indirection is assumed. The contents of the named data quantity will be used as the address of data and not as data.

The internal data organization attributes (type II) are listed in Table 2. 2. 2. A named data quantity may or may not be assigned a type II attribute. If no attribute is assigned the named data quantity is assumed to represent a single data item. If named quantity is assigned the array attribute, it must also be assigned size and dimension values. This is done with an assign statement. A named data quantity which is assigned an array, or stack attribute , will also be assigned a type I attribute of either integer, floating, or single length character or single length decimal. Thus, named data quantities which are organized into an internal data organization of either an array or a stack must contain homogeneous data.

The linked list and tree attributes are handled differently from stack and array attributes. The two linked list attributes are linked list item (LLI) and linked list head (LLH). The linked list item attribute is assigned to a named quantity which may have another type II attribute. The LLI attribute reserves additional storage

within the named data quantity to be used for linked list pointers. If the named quantity is inserted into more than one linked list, LLI(N) may be written and space for N linked list pointers is saved. Thus each linked list item must contain information relating where its pointer information ceases and its data begins. Also, if an item is linked in more than one list, and is removed from one of the linked lists, its position in the other lists is not affected. Thus a linked list with multiple pointers will incur more overhead than a singly linked list. The linked list head (LLH) attribute is assigned to a named data quantity which is to be the head of a linked list. The named data quantity with a LLH attribute can have other attributes including a LLI attribute. Thus a named data quantity could be the head of one list, a member of another list and also contain data. The manner in which linked lists are manipulated is explained in section 2.3.

The tree attributes are tree node and tree root. The trees are assumed to be binary trees. The tree root attribute is used to indicate that a named data quantity is the root of a tree. The named data quantity can also have other attributes. The root name of a tree is used in some tree operations and therefore requires additional storage. The tree node attribute causes the assignment of additional storage within the named data quantity to be used for the pointer required by a binary tree. A data item can be a node of more than one

tree, and therefore the TN(N) notation may be used to reserve storage for insertion into N binary trees.

A data item may be referenced by either its name or by its group name and displacement. For example, a single data item which is named F1 would be referenced as F1 while the third data item of a vector (one dimensional array) named F2 would be referenced as F2(3) or F2(F1) if F1 had the attributes integer, single data item, and also had the value of 3.

The elements of linked lists or binary trees are accessed differently than arrays, or single data items. Each linked list, or binary array has an associated key (K) which contains the location of one of its elements. The choice of which element is addressed by the key is controlled by linked list or binary tree operations (Section 2.3). An element of a linked list LL1 would be referenced as LL1 (K) if the element were a single data item. If the element were an array, the third member of the array would be referenced as LL1 (K, 3). It is possible to transfer the location (K) of an element into a named data quantity with attribute DL. In this case the item would be addressed using indirection.

In place of named quantities it is possible to use constants. A constant in a PTL statement is indicated by the constant value placed inside quotation marks. The attributes of the constant values can be assigned in two ways. The attribute can be specified explicitly,

immediately after the first quotation mark and separated from the constant value by a comma. If the attribute is not explicitly defined, it is implied by the constant value. If the constant value contains only numeric values it is assumed to be a base 10 integer constant. If the constant value contains only numeric values and a single period, the constant value is assumed to be a base 10 floating point constant. All other constant values which are without explicit attribute definition are assumed to be character strings.

To eliminate the need for the declaration of temporary storage areas and to provide the compiler with the knowledge that a value is required temporarily, temporary storage names are provided. The temporary storage areas are named !T1 through !TN. Temporary data storages are used to store temporary and intermediate results and have no explicit attributes. The attributes of a temporary data store match the data currently in storage.

The special named quantity !DUMP is used whenever an operation statement requires an information sink (named quantity) but the logic of the program does not require the data to be saved. The attributes of !DUMP match the data to be dumped into it. Thus, !DUMP acts as an infinite garbage can.

Data Location DL

Data location, address of data is stored in the named data quantity.

Character String CS(N)

Character representation; maximum number of characters in the string is N.

Binary String BS(N)

Binary string representation; maximum number of bits in the string is N.

Integer I(N)

Fixed point data representation; maximum length decimal digit to be represented has length of N.

Complex C(N)

Complex floating point data representation; maximum length precision is N decimal digits.

Floating Point F(N,M)

Floating point data representation; maximum length precision is N decimal digits; maximum range is $10^{M}$ to 10-M.

Decimal D(N)

Decimal representation; maximum number of decimal digits in N.

Table 2.2.1. Internal Data Representation Attributes

Type I

Array $A(i_1,i_2,i_3,\cdots i_N)$

A N dimensional array with 1-origin indexing is defined. The length of the 1st dimension is $i_1$, the 2nd is $i_2$, etc.

Stack S(N)

A push down stack (LIFO) is defined: the maximum capacity of the stack is N items.

Link List Head LLH

The head of a linked list is defined.

Link List Item LLI(N)

Space is provided so the named data quantity may be inserted into N lists.

Tree Root TR

The root of a binary tree is defined.

Tree Node TN(N)

Space is provided so the named data quantity may be inserted into N binary trees.

Table 2. 2. 2. Internal Data Organization Attributes

Type II

## 2.3 Program Type Language Statements

There are two basic statement types in the Program Type Language (PTL). The first statement type is the compile statement which is used to supply the microcompiler with information (data type, array size, etc.) needed to compile the operational statements. The second is the operational statement which is used to specify the CPU control and microprogram sequencing information. The compile statements will be presented first, then the operation statements will be given.

The compile statements are used to supply information which is required to interpret the operation statements. The two major functions of the compile statements are the description of named quantities and the specification of data storage requirements. In addition, the initial value of a named quantity may be specified using compile statements.

The dimensions of a named quantity must be specified concurrently with any attribute specification. The dimension may be specified implicitly, explicitly, or require a combination of both. For example, the length given a named data quantity possessing the character attribute explicitly specifies the required storage. However, a named data quantity possessing the attributes of array and floating point will require both an implicit and an explicit declaration. The size and dimension declaration explicitly specifies the number of floating point values which

must be stored. The precision declaration would implicitly imply either a single or double precision floating point specification. The variable would be assigned to whichever precision was greater than or equal to its specified precision. If the requested precision is greater than that provided by the hardware, the required precision will be implemented using special routines. It would be reasonable to flag the occurrence of such "large" precision requirements to notify the user of the inefficiencies in execution speed which are being introduced.

The specification of dimension, precision, or any other numerical value associated with an attribute of a named quantity follows the appropriate attribute (section 2.2) and is enclosed in parentheses.

The named data quantities may have as a NAME, any combination of 10 or less alphanumeric characters. The first character of a NAME must be an alphabetic character.

The compile statements are listed below. Examples will be given after each statement type to clarify the explanation.

2.3.0a BEGIN NAME

This declares the beginning of a main program with name NAME.

2.3.0b DECLARE SUB NAME (LIST)

This declares a subprogram named NAME. LIST is a list of parameters which will be passed to the subroutine by the calling program.

2.3.0c END SUB

This declares the end of a subroutine.

2.3.0d END PROG

This declares the end of a program.

2.3.1 DECLARE NAME ATTRIBUTES

A named data quantity with attributes and name as specified is created.

Examples

a) Create an integer vector named L1. The length of the vector is to be 10, and each element of L1 is to have a precision of at least 5 decimal digits.

DECLARE L1 A(10), I(5)

b) Create a 10 by 10 matrix of floating point numbers. Each number is to have a precision of 10 decimal digits, and the name of the matrix is to be MATR.

DEC MATR A(10,10), F(10)

c) Create a character string of length 10 and name CH1

DEC CH1 CS(10)

d) Create a linked list named LIST1. LIST1 is to serve as the head of the list and contain no other information.

DEC LIST1 LLH

e) Create a linked list named LIST2. LIST2 may be attached to 2 other linked lists, and is to contain 4 floating point numbers of precision 5.

DEC LIST2 LLH, LLI(2), A(4), F(5)

### 2.3.2 DECLARE NAMEFORM ATTRIBUTES

A group of named quantities is assigned the same attributes. Instead of listing the names, the NAMEFORM specifies the general class of names which are to be assigned the specified attributes when encountered within the program. NAMEFORM is a set of alphanumeric groups enclosed in parentheses. Each group starts with an alphabetic character and may have up to 10 alphanumeric characters. Any NAME which starts with one of the groups of the NAMEFORM set is assigned the specified attributes of the corresponding DECLARE statement.

Examples

a) All named data quantities starting with A1, A2, AB, or D are to be single valued integer quantities with a precision of 10 decimal digits.

DEC (A1,A2, AB,D) I(10)

b) All named data quantities starting with X are to be floating point vectors of length 6 with a precision of 5 decimal digits.

DEC (X) A(6) F(5)

2.3.3 VALUE NAME = $N_1(V_1),\cdots,N_m(V_m)$

Data is assigned to a named data quantity. If the named data quantity is a single item only $V_1$ is specified. If multiple items are required (NAME is an array) $V_1$ through $V_m$ are specified. If any $V_i$ is to be assigned to K consecutive position in storage, then $N_i$ is set to K, otherwise $N_i$ is omitted. The data is read from left to right, and the right-most array dimension changes most rapidly.

Examples

a) L1 of example 2.41a is to be assigned all 1's

VALUE L1 10(1)

b) MATR of example 2.41b is to be set to all 0's

VALUE MATR 100(0.0)

c) MATR (7,1) through MATR (7,5) is to be set to 1.0, 2.0, 3.0, 4.0, 5.0, respectively.

VALUES MATR (7,1) (1.0, 2.0, 3.0, 4.0, 5.0)

The general form of the operation statement is shown below (2.3.4).

2.3.4 LABEL OPERATION BRANCH

The three components of the operation statement are:

LABEL: The LABEL (1 to 10 alphanumeric characters with no intermittant blanks) is optional. If present, the LABEL is used to identify the operation statement. Any reference to an operation statement of a program (in particular a branch to an operation statement) is made using the LABEL identifier.

OPERATION: The OPERATION is optional, but either the OPERATION or the BRANCH must be present. The OPERATION specifies the operation to be performed by the CPU. The form of the OPERATION depends upon the OPERATION type. The OPERATION types will be listed below.

BRANCH: The BRANCH is optional, but either the OPERATION or the BRANCH must be present. The BRANCH can cause a break in the sequential execution of control statements. The form of the BRANCH depends upon the BRANCH type. The BRANCH types will be listed after the OPERATION types.

The set of OPERATION is divided into general categories such as arithmetic, bit manipulation, etc. Each category is further sub-divided into OPERATION subsets. An OPERATION subset may have

1 or more elements. The elements of each OPERATION subset have the same format and operate on the same data types.

To aid in the understanding of the OPERATION a detailed discussion of OPERATION (S1← S2 $\gamma$ S3)(2.3.5a) will be given. OPERATION (S1←S2$\gamma$ S3) is the first OPERATION subset and is part of the general category BIT MANIPULATION. The information is presented as follows. First the format of the OPERATION statement is shown. The OPERATION specified in 2.3.5a has several elements in the OPERATION subset and therefore the OPERATION identifier is replaced by $\gamma$.

The list of OPERATIONS of the OPERATION subset (S1←S2$\gamma$S3) is listed on the same line as the format. If any of the OPERATIONS requires additional explanation, the information is given on the lines below the format statement. The named quantities in the format statement are represented by S1, S2, and S3. In an actual statement this item would be replaced by named data. The use of different data items in the format (S1←S2$\gamma$S3) is not meant to imply that S1, S2, and S3 could not be the same named quantity. In fact all three data items in the format (S1, S2, S3) could be replaced by the same variable.

At the end of the format line are subsets of the attributes of tables 2.2.1 and 2.2.2. They indicate which types of data items the OPERATION statement may have as operands. The list of attributes is not included if the choice is obvious.

Many of the PTL functional operations have one or more associated condition sets. Condition sets are two-state devices which are set or reset dependent upon the results of PTL operations. The sequencing of PTL programs can be conditioned upon the state of one or more condition sets.

The setting of PTL condition sets can be handled in two ways. The programmer can be forced to indicate, in the PTL statement, that the result of an operation should modify the appropriate condition sets, or the language can be defined so that each time a PTL operation is executed, the associated condition sets are appropriately modified. The second method was chosen for PTL.

The automatic control of condition sets relieves the programmer of one more detail. The language is cleaner and more easily read without the notation required to specify the control of conditions sets. Therefore, the automatic control of condition sets should simplify the generation of PTL programs. There are some potential problems. The programmer should be cognizant of condition sets. The automatic control of condition sets tends to push them into the background. However, condition sets are specified directly in any branch statement which uses one or more condition sets.

A branch can be conditioned only upon the current state of a condition set. Using a PTL logical operation the status could be saved in a logical variable and tested later. However, the necessity to perform branching upon the previous state or states of a condition set would seem to occur infrequently.

For OPERATION (S1 ← S2 $\gamma$ S3), the condition sets are delayed, i.e., the state of the condition set switches are not changed until the end of the cycle. This has two effects. First, a branch can be made on the previous state of the condition sets during the present OPERATION statement. Second, a branch cannot be taken until the following microcycle. There are certain OPERATION types for which a branch can occur dependent upon the results during the same cycle (or at least appear to be dependent upon the current OPERATION)! These cases will be explained when they occur.

OPERATION SUBSETS

2.3.5 General category bit manipulation and logical operations

2.3.5a S1 ← S2 $\gamma$ S3 $\gamma \in [\Lambda, V, \Theta,$ NAND, NOR] B, I, F

condition set ZERO, ALL ONES

The named quantities S2, S3 are logically combined using the specified operation and the result is placed in named quantity S1. The condition set ZERO is set if the result is all zeros. The condition set ALL ONES is set if the result is all 1's.

2.3.5b S1← βS2 $\beta \epsilon$ [NOT] B, I, F

condition set ZERO, ALL ONES

The complement of S2 is placed in S1. If the result is zero, ZERO is set. If the result is all 1's ALL ONES is set.

2.3.5c S1←S2 TR ADDRESS S3 LABEL (S3;A)(S2;I)(S1:S3;I,F,CS,D)

condition set OUT BOUNDS

The Nth item of Table S3 is placed in S1, where N is the integer value of S2. If N is larger than the length of Table S3 a branch to LABEL is executed.

2.3.5d S1←S2 TR CONTENT S3 S4 LABEL (S3: S4; A)

condition set NO MATCH

A translation of S2 is made by matching its contents with Table S3. If a match is found the contents of the corresponding location of S4 is transferred into S1. The data attributes of (S2, S3) and (S1, S4) are required to be the same. If no match between S2 and Table S3 is found a branch to LABEL is executed.

2.3.5e S1←SHIFT S2 $\gamma_1\gamma_2 S_3$ (S3: I)(S1: S2; B,I, F)

$\gamma_1 \epsilon$ [LEFT, RIGHT] $\gamma_2 \epsilon$ [END AROUND, CARRY OUT, ARITHMETIC]

condition set ZERO, ALL ONES

The contents of S2 are shifted S3 places as specified by $\gamma_1\gamma_2$ and then stored in S1.

2.3.5f S1←S2←SHIFT S2$\gamma_1\gamma_2$S3 (S3; I)(S1:S2; B,I,F)

$\gamma_1 \epsilon$ [LEFT, RIGHT] $\gamma_2 \epsilon$ [END AROUND, CARRY OUT, ARITHMETIC]

conditions set ZERO, ALL ONES

Same as 5e, except that S2 is itself modified.

| | | |
|---|---|---|
| 2.3.6 | General category | arithmetic conditions |
| 2.3.6a | S1←S2$\gamma_1$S3 | $\gamma_1 \epsilon$[+, -, *, /] F,C,D |
| | conditions sets | ZERO, LESS ZERO, GREATER ZERO |
| | | OVERFLOW, UNDERFLOW |
| 2.3.6b | S1←$\gamma_2$S3 | $\gamma_2 \epsilon$ [$e^{S3}$, $\log_e$S3, $10^{S3}$, $\log_{10}$S3] F |
| | conditions sets | OVERFLOW, UNDERFLOW |
| 2.3.6c | S1←S2$\gamma_3$S3 | $\gamma_3 \epsilon$ [$S2^{S3}$] I,F |
| | conditions set | OVERFLOW, UNDERFLOW |

2.3.6d $S1 \leftarrow \gamma_4 S2$ $\gamma_4 \in$ [sin, cos, arc sin, arc cos, tan, cot, arc tan, arc cot]

conditions sets ZERO, LESS ZERO, GREATER ZERO, UNDERFLOW, OVERFLOW

2.3.6e $S1 \leftarrow \gamma_5 S2$ $\gamma_5 \in$ [L, ⌈ , ABS] I, F

conditions set ZERO, LESS ZERO, GREATER ZERO

The symbols L, ⌈ represent the floor and ceiling of S2 respectively. ABS produces the absolute value.

2.3.6f FLOAT S1 S2

Named data quantity S1 (with attribute I) is converted to an internal floating representation and placed into named data quantity S2 (with attribute F). S1 is not modified. S1 and S2 can not be the same named data quantity.

2.3.6g INTEGER S1 S2

condition set OVERFLOW

Named data quantity S1 (with attribute F) is converted to an internal integer representation and placed into named data quantity S2 (with attribute I). S1 is not modified. If S1 is too large to be contained in S2 the condition set overflow is set. S1 and S2 can not be the same named data quantity.

2.3.7 General category stack operations

Several of the stack operation types carry a branch LABEL as part of their format. These operation types have an implied branch which is tested before the execution of the desired stack operation occurs. If the branch condition is matched the branch occurs and the operation is not performed. The conditions being tested represent the state of the stack after completion of the last stack operation.

There is one common stack (ST) which does not have to be set up, but is always available.

2.3.7a S1←ST LABEL I, F, C, D

conditions set STACK EMPTY, STACK FULL

Branch to LABEL if STACK EMPTY, otherwise place top item of STACK in S1.

2.3.7b ST←S1 LABEL I, F, C, D

conditions set STACK EMPTY, STACK FULL

Branch to LABEL if STACK FULL, otherwise place data item S1 into STACK

2.3.7c CLEAR ST

conditions set STACK EMPTY, STACK FULL

2.3.7d CREATE ST (S1) S2$\gamma$ $\gamma \in$ [I, F, C, D]

conditions set STACK EMPTY, STACK FULL

A STACK of type $\gamma$ with name S1 and length S2 is created.

2.3.7e DESTROY ST(S1)

A stack named S1 is destroyed

2.3.7f S2←ST(S1) LABEL

conditions set STACK EMPTY, STACK FULL

If STACK S1 is empty a branch to Label occurs, otherwise top item of STACK S1 is placed in S2.

2.3.7g ST(S1)←S2 LABEL

conditions set STACK EMPTY, STACK FULL

If STACK S1 is full a branch to LABEL occurs, otherwise S2 is placed into STACK S1.

2.3.7h CLEAR ST (S1)

conditions set STACK EMPTY, STACK FULL

2.3.8 General category linked lists

A few general points should be made before presenting the linked list operations. Each linked list has a key associated with it. The key contains the address of some member of the linked list unless the list is null. Most of the linked list operations use the value of the key, modify the key, or transfer the value of the key into a named

data quantity.

Each element of a linked list must contain space for a linking area. The linking area contains pointers to the next and to the preceding element of the linked list. (Some implementations might use other schemes.) A named data quantity may be a member of more than one list. The attributes LLI(N) specifies the number of linked lists (N) which may contain the named data quantity simultaneously. The linking area size is a function of N. The insertion or deletion of a data item from one list does not affect its membership in any other linked list. The linking area also contains information to delineate the end of the linking area and the beginning of the data area.

Most of the linked list operations include a LABEL and have an associated group of condition sets. A branch to LABEL is executed during the time span of the linked list operation if any of the branch conditions occur. The state of the condition sets (after the branch) can be tested to determine which condition caused the branch. A branch statement can be contained on the same line as a linked list operation. However, any branch caused by the linked list operation will have precedence over the branch

statement. Thus, a branch to LABEL 1 on the result of a previous add operation being zero could be placed in the same line as a linked list operation which branched to LABEL 2 if the linked list were empty. If the linked list was not empty and the result of the previous add was zero; a branch to LABEL 1 would result. If the list was empty and the result of the previous add was zero; a branch to LABEL 2 would result.

2.3.8a SET LINK LIST S1

Named data quantity S1 (having attribute LLH) has its linked list key initialized.

2.3.8b DESTROY LINK LIST S1

Named data quantity S1 (having attribute LLH) has its linked list key initialized , and the elements of the linked list have their associated pointers reinitialized.

2.3.8c REMOVE S1 LINK LIST S2 $\beta$ LABEL; $\beta \in [+1, 0, -1]$

condition set OUT BOUNDS

The element of linked list S2 addressed by the key ($\beta$=0) or the preceding element ($\beta$= -1) or the next element ($\beta$= +1) is removed from the linked list. The element's linking area is reinitialized and the address of the element is

placed in S1. The key is updated to point to the element preceding the removed element. If the key with the $\beta$ offset points outside the list a branch to label is executed.

2.3.8d S1←LINK LIST S2 $\beta$ LABEL $\beta \in [+1, 0, -1]$

condition set OUT BOUNDS

The address of the element of link list S2 addressed by the key ( $\beta$=0), or the preceding element ($\beta$= - 1), or the next element ($\beta$=+1) is placed into S1. If the key with the $\beta$ offset points outside the linked list a branch to LABEL is executed.

2.3.8e REPLACE S1 IN LINK LIST S2 $\beta$ LABEL $\beta \in [+1,0,-1]$

condition set OUT BOUNDS

The data item pointed to by the key with an offset of either +1, 0, or -1 is removed and replaced by data item S1. The key is set to the value of the key with offset. If the key with offset points outside the linked list, a branch to LABEL is executed.

2.3.8f INSERT S1 IN LINK LIST S2 $\beta \in [+1, -1]$

Data item S1 is placed in linked list S1 either before ( $\beta$=-1) or after ($\beta$=+1) the data item pointed to by the key. The key is updated to point to the inserted data item.

2.3.8g INCREMENT POINTER LINK LIST S1 LABEL

condition set OUT BOUNDS

The key associated with linked list S1 is incremented by 1

2.3.8h DECREMENT POINTER LINK LIST S1 LABEL

condition set OUT BOUNDS

The key associated with linked set S1 is decremented by 1

2.3.8i KEY (S1)←S 2 LINK LIST LABEL

condition set OUT BOUNDS

The key is updated to point to the Nth item of linked list S1 where N is the value of S2. If the number of elements of S1 is less than S2 a branch to LABEL is executed.

2.3.8j S2←KEY (S1) LINK LIST

The key associated with linked list S1 points to the Nth element of list S1. The value of N is placed into S2.

2.3.8k DO LABEL LINK LIST S1

The statements from the DO statement to the statement LABEL are iterated N times where N is the number of elements in linked list S1. The key is set for the first element on the first iteration, to the second element on the second iteration, etc.

2.3.8l DO LABEL LINK LIST KEY S1

Same as 2.4.8k except the iteration is done for the current value of the key to the last element of the list.

2.3.9 General category string manipulation

2.3.9a CONCATENATE S1 to S2

String S1 is concatenated to string S2

| | |
|---|---|
| EXAMPLE | S1 = 'AB' |
| | S2 = 'CD' |
| after | CONC S1 to S2 |
| execution | S2 = 'CDAB' |

2.3.9b MATCH S1 IN S2 S3 LABEL

A pattern match search in string S2 using string S1 is performed. If a match is found S3 contains the position of the first character of the match. The search is performed from left to right. If no match is found a branch to LABEL is executed.

| | |
|---|---|
| EXAMPLE | S1 'ABC' |
| | S2 '123ABCD' |
| after | MATCH S1 IN S2 S3 LABEL |
| execution | S3 = 4 |

2.3.9c MATCH P S1 IN S2 S3 LABEL

This operation is the same as 2.3.9b except that the search is started in S2 at the character position specified by S3. If no match is found, or S3 points outside S2 a branch to LABEL is executed.

EXAMPLE S1 = 'AB'

S2 = 'ABCDABF'

S3 = 2

after MATCH P S1 in S2 S3 LABEL

execution S3 = 5

2.3.9d INSERT S1 IN S2 AT S3 LABEL

String S1 is inserted into string S2 after the character position specified by S3. If length of S1 and S2 exceeds the maximum length of S2 a branch to LABLE is executed.

EXAMPLE S1 = 'BC'

S2 = 'ADEF'

S3 = 1

after INS S1 S2 S3 LABEL

execution S2 = 'A B C D E F'

2.3.9e GET S4 FROM S1 AT S2 THROUGH S3 LABEL

The string of characters in string S1 between the character positions specified by S2 and S3 is placed into string S4. If S2 or S3 points outside of S1 a branch to LABEL is executed.

EXAMPLE S1 = 'ABCD'

S2 = 2

S3 = 4

after GET S4 S1 S2 S3

execution S4 = 'BCD'

S1 = 'ABCD'

2.3.9f DELETE S2 THROUGH S3 FROM S1 LABEL

The string of characters between the character positions specified by S2 and S3 is deleted from string S1. If S2 or S3 points outside of S1 a branch to LABEL is executed.

EXAMPLE S1 = 'ABCD'

S2 = 2

S3 = 4

after DEL S2 S3 S1

execution S1 = 'A'

2.3.9g REPLACE BY S4 BETWEEN S2 AND S3 IN S1 LABEL

The string S4 replaces the characters in string S1 between S2 and S3. If S2 or S3 points outside S1 a branch to LABEL is taken.

EXAMPLE S1 = 'ABCDE'

S2 = 3

S3 = 5

S4 = '2'

after REP S4 S2 S3 S1 LABEL

execution S1 = 'AB2'

2.3.9h S1←STRING LENGTH S2

The length of string S2 is placed in S1.

2.3.10 General category - iteration control

2.3.10a DO LABEL S1 I

The statements from the 'DO' statement to the statement LABEL are iterated S1 times. The iterations can be terminated by a branch out of the scope of the DO.

2.310b ADO LABEL ARRAY S0 INDICES S1, S2, ···SN ORDER

The statements from the 'ADO' statement to the statement LABEL are iterated. The named quantities specified by S1, S2, ···SN are treated as the indices of array S0, with S1

the first index (the right most), etc. The indices specified by ORDER (which is a named data quantity) are counted from 1 to the dimension of the indices. The counting is done so that all elements of array S0 which may be specified by the selected indices are counted. ORDER is treated as a hexidecimal variable. The lower order hexidecimal digit of ORDER specifies the index which is counted most rapidly. The second hexadecimal digit specifies the index which is counted the second most rapidly, etc.

EXAMPLE ADO LABEL1 AR A1 IN S1,S2,S3,OR1

Assume A1 has dimensions 2,3,4 and OR1 is $31_6$.

Then S1, S2, S3 would be counted as follows

| | | | |
|---|---|---|---|
| S3 | S2 | S1 | start iteration |
| 1 | S2 | 1 | |
| 1 | S2 | 2 | |
| 1 | S2 | 3 | |
| 1 | S2 | 4 | |
| 2 | S2 | 1 | |
| 2 | S2 | 2 | |
| 2 | S2 | 3 | |
| 2 | S2 | 4 | end iteration |

Notice that the value of S2 is unchanged during the execution of the loop instruction.

2.3.10c ADO1 LABEL ARRAY S0 INDICES S1,S2,···SN ORDER I

This operation statement is the same as 2.3.10b except that none of the indices are set to 1. The indices are counted from their current value.

2.3.10d IDO LABEL S1 S2 $\alpha$ (S2: I)(S1: A)

$1 \leq \alpha \leq$ (DIMENSION ARRAY S1)

The statements from the 'IDO' statement to the LABEL statement are iterated. The named quantity S2 is set for the first iteration and incremented by 1 before each of the succeeding iterations. The named quantity S1 is an array of order N with a specified length for each of the N dimensions. The value of $\alpha$ selects which dimension controls the iteration. If $\alpha$ is i and the length of the ith dimension of S1 is j then j iterations will be executed.

EXAMPLE Assume A1 has order 4 with dimension length (3,4,2,5). Then IDO LABEL A1 S2 '3' will cause 2 iterations to occur.

IDO LABEL A1 S2 '1' will cause 3 iterations to occur.

2.3.11 General category binary trees

A few general comments should be made before

presenting the binary tree operations. Each binary tree has a key associated with itself. The key contains the address of some member of the tree unless the tree is null. Most of the binary tree operations use the value of the key or modify the key.

Each item inserted into a binary tree must provide space for linking information. The space is provided by assigning each named quantity which is to be inserted into binary tree the TN attribute. The binary tree operations are involved with inserting, deleting, and sequencing through binary trees for the location of the data items of the tree. Thus, addresses are the result of binary tree operations. The address specifies the location of some member of the tree.

2.3.11a SET TREE ROOT S1

Named data quantity S1 (having attribute TR) has its binary tree key initialized.

2.3.11b DESTROY TREE ROOT S1

Binary Tree S1 (having attribute TR) has its binary tree key reinitialized and any elements (nodes) of tree S1 have their tree linkage areas reinitialized.

2.311c MOVE TREE KEY S1 $\alpha$ LABEL

$\alpha \in$ [UP, DOWN LEFT, DOWN RIGHT]

condition set OUT BOUNDS

Binary Tree S1 has its associated key moved UP, DOWN LEFT, or DOWN RIGHT if $\alpha$ =UP, DL, or DR. The key contains the address of the corresponding node of tree S1. If the command would cause the key to move off the tree (the corresponding node does not exist) a branch to LABEL is executed.

2.3.11d MOVE TREE KEY S1 ROOT

The key associated with binary tree S1 is reset to point to the root of the tree (S1).

2.3.11e REMOVE TREE S1 $\alpha$ LABEL

$\alpha \in$ (UP, CURRENT, DOWN LEFT, DOWN RIGHT)

condition set NO NODE, BINARY VIOLATION

The node of tree S1 relative (UP, CURRENT, DL, DR) to the position of the key is removed from the tree. The data quantity removed from the tree has its tree linkage area reset.

If a node does not exist at the position specified by $\alpha$ relative to the key, a branch to LABEL is executed and condition set NN is set. If removal of the node and the subsequent regrouping of the tree would produce a non-binary tree, the node is not removed, a branch to LABEL is executed and condition set BV is set.

2.3.11f REPLACE TREE S1 $\alpha$ BY S2 LABEL

$\alpha \in$ (UP, CURRENT, DOWN LEFT, DOWN RIGHT)

condition set NO NODE

The node of tree S1 relative (UP,NODE,DL,DR) to the position of the key is replaced by named data quantity S2. The data quantity removed from the tree has its linkage area reset. If a leaf does not exist at the pointer specified by $\alpha$ relative to the key a branch to LABEL is executed.

2.3.11g INSERT TREE S1 $\alpha$ S2 LABEL

$\alpha$ (UP, DOWN LEFT, DOWN RIGHT)

condition set NO LEAF

Named data quantity S2 is inserted into the binary tree S1. S2 is inserted between the node currently addressed by the key and the node specified by $\alpha$ relative to the key. The orientation of nodes is kept constant. If S2 is inserted

between two nodes with orientation down right the three nodes will have orientation down right. If there is no node position (pointer + $\alpha$) where $\alpha$ is DL or DR, S2 is simply added to the tree at that position. If $\alpha$ is UP and no node exists, a branch to LABEL is executed.

2.3.11h Do Label Tree S1 $\alpha$

$\alpha \in$ (PREORDER, POSTORDER, ENDORDER)

The statement from the DO statement to the statement LABEL are iterated for N times where N is the number of nodes in the tree. The pointer is set to point to each leaf of the tree in either preorder, post order, or endorder according to $\alpha$.

2.3.11i S2←TREE KEY (S1)

The address of the node of tree S1 currently contained in tree S1's key is transferred to S2.

2.3.12 General category input/output

The I/O instructions represent a large set of operations. For example, the I/O READ instruction may be an I/O READ TAPE, I/O READ DRUM, or I/O READ DATA LINE, etc. operation. It is doubtful if CPU hardware could be provided which would perform the operations in one micro-cycle or less. Therefore, each I/O instruction in a PTL

program will probably be replaced by a branch to a large I/O subroutine.

2.3.12a SENSE I/O S1 S2

Status information from the I/O device specified by S1 is stored in location S2.

2.3.12b COMMAND I/O S1 S2

Control information located in area S2 is transmitted to the I/O device specified by S1. If S2 is enclosed in quotes the S2 name represents a standard command which is known to the translator.

2.3.12c READ I/O S1 S2 S3

Data from I/O device S1 is read into area S2. Location S3 is optional, but if present is used to store status information at the end of the read operation. Location S3 is set to null status at the beginning of the read operation.

2.3.12d WRITE I/O S1 S2 S3

Data from area S2 is written into the I/O device specified by S1. Location S3 is optional, but if present is used to store status information at the end of the write operation. Location S3 is set to null status at the beginning of the write operation.

2.3.12e CONVERT OUTPUT S1 FORMAT S2

The single data quantity S1 is converted to an output form according to format. The output form is stored in S2.

| | | |
|---|---|---|
| FORMAT: | 1,N | INTEGER to N place BINARY |
| | 2,N | INTEGER to N place OCTAL |
| | 3,N | INTEGER to N place DECIMAL |
| | 4,N | INTEGER to N place HEXADECIMAL |
| | 5,N | DECIMAL to N place DECIMAL |
| | 6,N | DECIMAL to N place DOLLARS, CENTS |
| | 7,N | FLOATING POINT to N place INTEGER |
| | 8,N,M | FLOATING POINT to N place DECIMAL with M significant digits |
| | 9,N,M | FLOATING POINT to N place SCIENTIFIC with M significant digits. |

2.3.12f CONVERT INPUT S1 FORMAT S2, S1 $\epsilon$ C S2 $\epsilon$ I, F1, F2, D

The character string S1 is converted into an internal data form according to FORMAT. The data item is stored in S2.

| | | | |
|---|---|---|---|
| FORMAT: | 1 | DOLLAR, CENT | to DECIMAL |
| | 2 | DECIMAL | to DECIMAL |
| | 3 | BINARY | to INTEGER |
| | 4 | OCTAL | to INTEGER |
| | 5 | DECIMAL | to INTEGER |
| | 6 | SCIENTIFIC | to FLOATING |
| | 7 | DECIMAL | to FLOATING |

2.3.13 General category - Special Functions

2.3.13a INCR S1

Named data quantity S1 (with attribute I) in incremented by 1

2.3.13b DECR S1

Named quantity S1 (with attribute I) is decremented by 1

2.3.14 General Category - Interrupts

The processing of an interrupt is visualized as a three step process. The first step is to turn on the interrupt. Once an interrupt is turned on, the occurrence of the interrupt can be accepted by the CPU. The second step is the acceptance of the interrupt. An interrupt is accepted by being placed in the interrupt queue (I queue). The interrupts in the I que are ordered on a basis of priority. Each interrupt is assigned a priority number. The interrupt with the highest priority is placed first in the I queue. Elements

in the I queue with the same priority number are ordered on a first-in first-out basis. The third step is to acknowledge the interrupts. To acknowledge an interrupt the currently executing process (program) is stopped, and its status is stored. The stored status must be sufficient to restart the interrupted program without error. After the currently executing program is stopped and its status stored, a branch to the proper interrupt routine is taken. Once the interrupt is handled, the status of the previous executing program is restored and the program is restarted. Therefore, provision must be made to form a queue of interrupted program status information. The interrupted program queue is called the R queue. The R queue contains both the status of the interrupted program and the information about the interrupt condition which caused the interrupt.

2.3.14a SET INTERRUPTS ON S1,...,SN K LABEL

This instruction turns on the interrupts for device conditions S1,S2,...,SN. The interrupts specified by S1,...,SN may be a particular device interrupt condition, or an I/O device name, in which case, all the interrupts associated with the device are turned on. The value K assigns a priority to S1,...,SN. If one of the interrupts S1,...,SN is acknowledged a branch to LABEL occurs.

2.3.14b SET INTERRUPTS OFF S1,S2,...,SN

This instruction turns the interrupts off for device conditions S1,S2,...,SN. The interrupts specified by S1,...,SN may be a particular device interrupt, or an I/O device name in which case all the interrupts associated with the device are turned off.

2.3.14c HOLD INTERRUPTS K

All interrupts which occur with priority K or lower are not acknowledged but are held in a que for later processing. The interrupts are arranged in the que with those interrupts with the highest K first. Within the same K class the interrupts are ordered on a first in first out basis.

2.3.14d RELEASE INTERRUPTS K

All interrupts K or higher which are in the interrupt queue are released for acknowledgment. The queue is processed from the top of the queue. If a new interrupt occurs before the interrupt queue is empty, the new interrupt is placed in the queue and ordered with the highest K first. Within the same K class the interrupts are ordered on a first in first out basis.

2.3.14e EMPTY I QUEUE K

All interrupts in the I QUEUE with priority K or less are deleted.

2.3.14f INTERRUPT RETURN S1

If S1 is not present the top process in the interrupted process queue R queue is restarted. If S1 is present the top process in the R queue is deleted and a branch to S1 is taken.

2.3.14g GET INTERRUPT STATUS S1

The interrupt information of the last acknowledged interrupt (top of R queue) is placed in S1. The R queue is not affected.

2.3.15 GENERAL CATEGORY - BRANCH OPERATIONS

The branch types are few in number and rather basic. The power of the branching operations stems mainly from the condition sets. The condition sets in total allow an effective check upon the operations which have been performed by the CPU. The branch types are listed below.

BRANCH Types

2.3.15a BR LABEL $\gamma$ F(CS, $\Lambda$, V, -)

$\gamma$ is the identifier for some set of OPERATION types. F(CS, $\Lambda$, V, -) is a function of the CONDITION SETS associated with $\gamma$ and the operation of AND, OR, NEGATION. The branch to LABEL occurs if the present states of the condition sets cause F(CS, $\Lambda$, V, -) to be true.

2.3.15b BR LABEL S1 $\gamma$ S2; $\gamma \in [=, \leq, \leq, \geq, >]$

2.3.15c BR LABEL S1 $\gamma$ C; $\gamma \in [=, \leq, \leq, \geq, >]$

C is a constant

2.3.15d BR LABEL I/O COND S1

A branch to LABEL if the I/O status which is stored in S1 agrees with COND.

2.3.15e BR SUB(NAME) (LIST)

A branch to the subroutines named NAME is executed. LIST is a set of parameters which must be passed between the subroutines and the calling program.

2.3.15f RETURN

A return from a subroutine to the calling program is made.

## 2.4 Differences Between PTL and Other Languages

We would like to compare and contrast PTL with both higher level languages and assembly level languages. Only one assembly level language will be used for comparison and that will be the IBM 360 assembly language. The purpose of the comparisons is to defend the decision to develop a new language instead of using an existing language. The comparison will indicate that PTL lies somewhere between higher level languages and assembly level languages in complexity, not equidistant, but closer to high level languages.

Let us compare PTL and IBM 360 assembly language first. What advantages does PTL have over the assembly language? First, the instruction set is much more powerful. The string manipulation, convert, stack, etc., operations are not present in the IBM 360 assembly language. Therefore, it should be easier to write PTL programs than assembly language programs if stack, list, etc., type operations are required. The PTL program should also be faster for these complex operations since direct implementations of complex operations in micro-code is usually faster than using several machine language instructions. At the very least, one level of interpretation is eliminated for these operations.

The PTL language is architecture independent since it does not refer to registers, local storage areas, or other machine dependent entities. A program written in an assembly language is architecture

dependent and can be run only on one machine or a limited class of machines unless the program is run under emulation or simulation. A program run under simulation always makes inefficient use of computer resources and generally runs at a slow rate. A program run under emulation tends to execute reasonably fast but makes less than optimal use of the computer resources.

It could be argued that a machine language could be written which would contain the PTL operations as its base. This approach would present some problems. Some PTL operations require as many as four operands. The operands can be located in registers or main storage and therefore, each instruction can have several format types. Thus, the instruction code length would necessarily be quite long to accommodate the many different instruction types. As a practical matter, it would seem reasonable to restrict the proliferation of format types by binding some of the operand locations to either main core or registers. The binding of operands can reduce the program execution speed. This occurs for two reasons. If an operand is in main core but should be in a register (or high speed local storage) for instruction execution, a main core memory cycle will be required. If such an instruction is executed several times within a short loop all its operands should be maintained in registers, but this could be impossible since the format binds some of the instructions operands to main core. The opposite

effect can also happen. It may occur that a sequence of instructions require more registers than are available. Therefore, instructions which move operands back and forth between main core and register storage will be required. The amount of operand movement will be increased if the formats restrict the location of some operands to registers.

The major inefficiency occurs because the machine language has to fix the location of operands and can not make efficient use of high speed local storage on a global basis. The use of local storage will be optimized only on an instruction by instruction basis. It should be pointed out (see Section 2.5) that the assignment of data quantities to storage locations is one of the major tasks of the PTL translator.

A comparison between PTL and other higher level languages will be made on a function by function basis. Most of the comparison will be made between PTL and Fortran with references to PL1, SNOBOL, APL, and others.

The equivalent of NAMELIST in FORTRAN or DATA in PL/1 would require several PTL statements. The PTL statements would implement the NAMELIST operation using string search, string get, and input convert operations.

The input output capabilities of APL and PTL can not be compared on an equitable basis. APL is an interpretive language with

rather limited I/O capabilities. More sophisticated I/O features could be added to APL if desired.

The testing and control of I/O devices is more complete in PTL than in FORTRAN and PL1. In PTL the program can specify exact I/O conditions to branch on and can send specific I/O control signals. (APL has no explicit I/O control so no comparison can be made.)

The arithmetic operations in PTL are individually very easy to use. The operands can be subscripted in a manner similar to FORTRAN. However, each arithmetic operation requires all the operands to have the same internal data representation. Thus any mode conversion must be specified by the programmer. In Fortran and PL1 mode conversion is accomplished automatically without any programmer effort. A difference between PTL and many of the higher level languages is the inability to express multiple arithmetic operation in one line. Thus in Fortran $x = (y * 2 * z)/\omega$ can be expressed in one line. In PTL the same expression requires three lines of code. This problem could be resolved with the use of a precompiler which converted FORTRAN-like arithmetic statements.

APL has the most sophisticated arithmetic operations of any language of which we are aware. The generalized matrix product, outer product, rotation, summation of vector components, etc.,

allow very powerful programs to be written in a few statements. The APL statements mentioned above can be simulated by using PTL statements. The simulation is made easier using the array iteration facilities of PTL. Fortran and PL1 would require more statements than would PTL to emulate those APL operations.

The iteration control in PTL is somewhat limited in scope since the index variable can only be incremented in steps of 1, and the index is always positive. The iteration for arrays and lists provided in PTL are very powerful and can be very useful.

The string manipulation operations of PTL are the basic set of concatenation, insert, delete, get, replace, and match. The more complicated string manipulation operations which are available in SNOBOL can be accomplished in PTL by combining several PTL operations. String manipulation is harder to do in PTL than in SNOBOL. This is to be expected since SNOBOL is a special purpose string manipulation language.

The list operations in PTL are basic but adequate to handle single list type operations. The operations one normally performs in lists are reasonably simple. The capability to search a list for a match of a particular subelement of each list element is provided by an iterate operation which controls a loop and iteratively selects the location of each list element. Whatever search is required is then done within the loop. A limited set of list operations is

provided in PL/1 but is not available in FORTRAN or SNOBOL.

The full complement of stack operations are provided in PTL. There are very few stack operations that are required anyway. However, stack operations are not provided in the other languages at all.

The branch operations in PTL cannot really be compared with those in higher level languages. The manner in which they are specified is quite different.

The comparison between PTL, 360 assembly language, and higher level languages has not been made to assert the superiority of one or the other of the languages. The comparison has been presented to place PTL in perspective relative to the other languages.

Why design a new language? What can't an existing language such as APL, FORTRAN, PL/1, or SNOBOL do as well as PTL?

First, the languages mentioned above and all other languages with which we are familiar mask the user's true intent. Thus, functional statistics gathered on programs written in these languages would not be valid. Such statistics would not provide for truly optimal hardware selection. For example, Fortran does not have list operations, bit manipulation operations, or string operations. PL/1 does not have binary tree operations or stack operations. The list operations which are provided in PL/1 are not sufficient for even minimum list manipulation. SNOBOL does not

have floating point arithmetic operation, APL does not have explicit floating point or integer arithmetic operations, PL/1 does not have sufficient stack, linked list, or binary tree operations. Thus, the languages are not amenable to the estimation of valid statistical information on functional usage.

Second, the languages would be difficult to compile into efficient microcode. In particular, APL and PL/1 have a large number of basic constructs. Thus, the number of in-line microcode subroutines which would have to be supplied to the compiler would be excessively large. Some of the constructs of PL1, APL, SNOBOL, and FORTRAN are so complicated that the microcode for such routines would be excessively large. Such routines might require more temporary storage and variable storage than can be supplied by local storage. This would imply that some storage would be assigned before translation and could result in inefficient code. The problem of storage assignment is explained more fully in Section 2.5.

Third, it seems reasonable to write compilers, interpreters, and other heavily used programs in just "one language". Then, as was mentioned earlier, only one new translation is required for each new computer system. PTL is designed to compile efficiently and contains stack operations and other operations which are used in writing interpreters and compilers. Thus PTL seems a good candidate for the "one language".

The preceding paragraphs have considered the languages (SNOBOL, PL1, APL, FORTRAN) as a group. It might be well to consider their limitations on an individual basis.

SNOBOL is a special purpose language applicable to only one problem area and it thus not a suitable candidate for a PTL-type language. FORTRAN is more general-purpose than SNOBOL. However, FORTRAN is very scientifically oriented and is missing a number of operations which are necessary for a PTL-type language. APL is a very general language but has several problems. The variables in APL require dynamic storage allocation since they can vary in size from instruction to instruction. The dynamic allocation of storage would add unnecessary and unwanted overhead to both the translation process and the execution process. The remaining objections to APL are the author's bias. I think that APL is too concise. It is possible to write APL code which is practically unreadable. Also, the concise notation carries over to branch conditions and can cause end conditions within the branching operations which are sometimes hard to foresee. PL1 is a very general language. As was pointed out earlier some desirable program types are not available in PL1; however, the biggest problem with PL1 would be the generation of efficient microcode. The ability to dynamically allocate memory would be a serious problem, reducing both the efficiency of the compiler and the efficiency of the final microcode. Just the massive

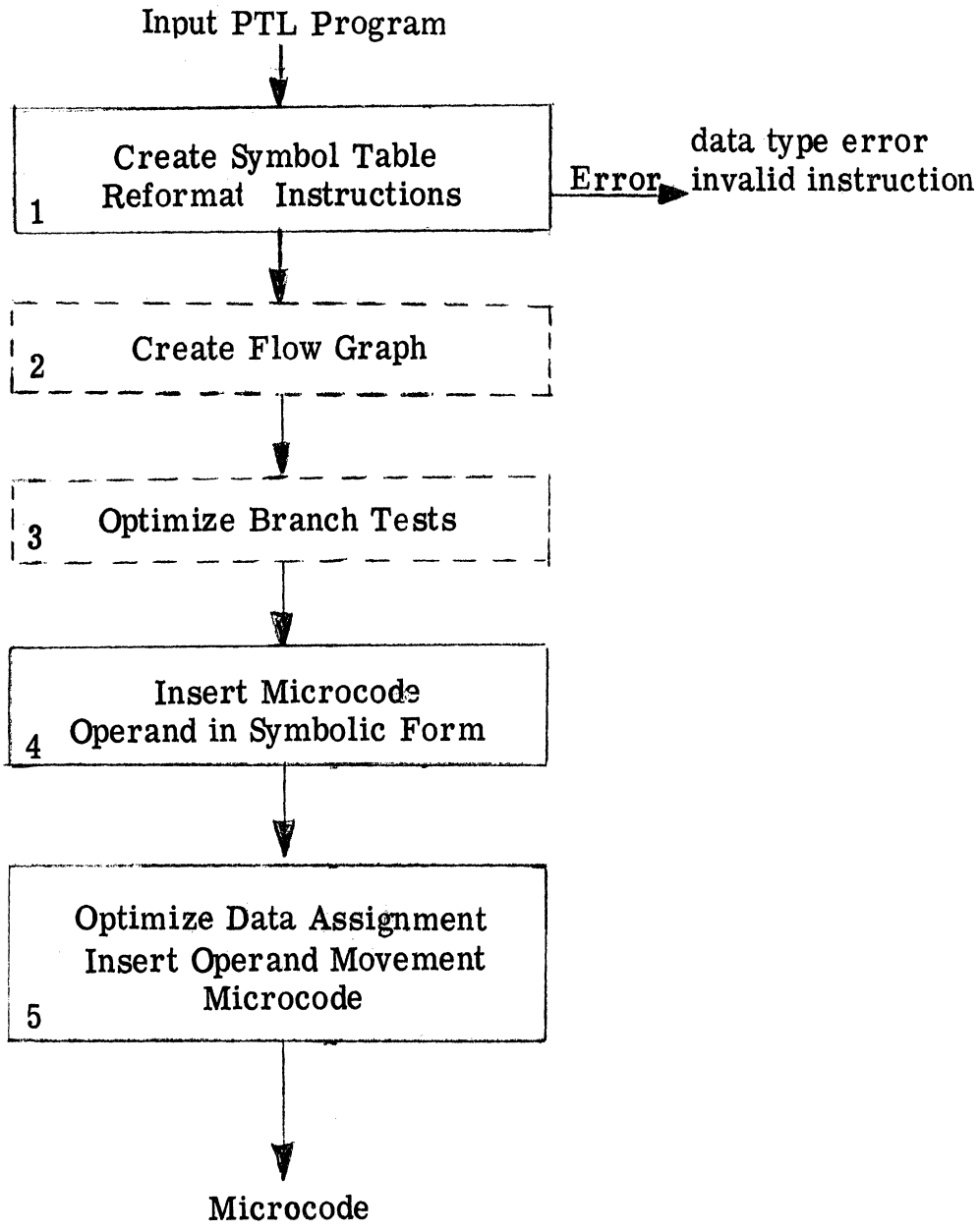
size of PL1 would make the generation of the PL1 program type implementations very difficult.

## 2.5 Translation of PTL Programs

The purpose of this section is not the design of a PTL translator. Instead, a block diagram (Figure 2.5.1) of a PTL microtranslator is presented. The block diagram represents the major steps required to translate a program written in PTL into microinstructions suitable for execution upon a microprogrammable CPU. The functions performed by each block of Figure 2.5.1 will be discussed below. In particular, those problems introduced by the use of microprogamming will be elaborated.

The first block is a scanner. It performs a standard set of operations. The symbol table is formed. Each program statement is placed in a standard form. The operation name is changed into an identifier code, and the operands are ordered with each operand replaced by its symbol table entry. If any of the operands are arrays, indexing variables and indexing constraints are noted. If a statement includes a branch operation, it is also placed into a standard form with the branch operation name replaced by an identifier code. Any errors, in data assignment, instruction labeling, or instruction formatting are flagged.

The operations performed by the first block are basically the same as those performed by a non-microprogram scanner analyzer.




Block Diagram of Process Required to Translate PTL Programs into Microcode

Figure 2.5.1

Therefore, the design of the scanner analyzer portion of the translation should present no new problems.

The second and third blocks of Figure 2.5.1 are used to minimize the overhead associated with the setting and resetting of condition sets (see Section 2.4.1).

If a CPU does not have hardware which checks and sets a flip flop or other two-state device for a desired condition, such hardware must be simulated by the microprogram. For example, the add operation has an associated condition set which is set if the result of the addition is all zero. If the hardware does not perform the test, then several microinstructions are required to set the condition set. In the worst case, the result of the addition must be compared with zero. Then the condition set word must be read out of storage, set or reset depending upon the result of the previous compare operation and then rewritten. If these additional microoperations are required for every add operation that appears in a PTL program and similar microoperations are required to handle other PTL operations, the bookkeeping associated with condition sets becomes very restrictive.

However, even though most PTL operations produce a condition set or condition sets, the status of the condition sets for that operation in that position in the program is never tested. The function of blocks two and three is to eliminate the modification of condition sets for operations whose results are not tested. If the setting of condition

sets requires no overhead, that is to say if the condition sets are handled by built-in hardware, there is no need to eliminate unused setting of condition sets because they do not result in the addition of microcycles. For such CPU's block 2 and block 3 would be eliminated from the microcompiler.

The function of block 2 is to produce a flow diagram of the program. This is simply a graph where each node of the graph is either a program label or a conditional branch. The arcs of the graphs are directed arcs and each arc represents a conditional branch from a branch operation to a program statement.

The function of block 3 is to determine which statements within the program require the modification of their associated condition sets. The method used to select the appropriate statements is quite simple. The program flow diagram produced by block 2 contains all conditional branches. For each conditional branch a search is made using both the PTL program (output from block 1) and the program flow diagram to determine which PTL operations could affect the appropriate condition set. Only those operations whose results are tested by a branch statement via a condition set will add microcode for the modification of the appropriate condition sets. Most PTL statements whose results are not being tested will not modify condition sets. The examples of Figures 2.3.2a and 2.3.2b indicate the function performed by blocks 2 and 3.

Figure 2.5.2

Assume the following operations

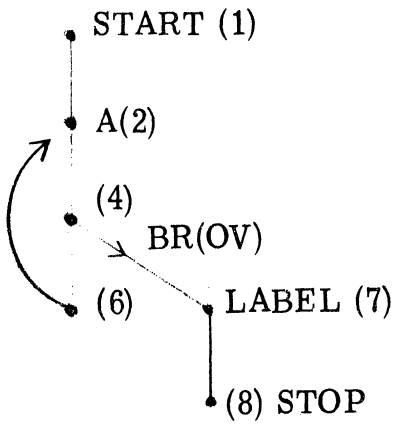
| | | |
|---|---|---|
| ADD | S1 ← S2 + S3 | |
| | Condition sets | ALL ZERO, OVERFLOW |
| SHIFT | S1 ← S2 SHIFT S3 | |
| | Condition sets | CARRY OUT |
| BRANCH | BR (Condition True) | LABEL |

Program 1

| Statement # | | |
|---|---|---|
| 1 | START | S1 ← S2 + S4 |
| 2 | A | S2 ← S2 SH S1 |
| 3 | | S1 ← S2+S3 |
| 4 | | S3 ← S2 SH S2, BR(OV)LABEL |
| 5 | | S1 ← S2 + S4 |
| 6 | | BR A |
| 7 | LABEL | S1 ← S2 + S4 |
| 8 | | STOP |

For this program, only the status of the add overflow condition set must be modified and need be modified only after execution of statement 3.

(a)




Function Performed by Blocks 2 and 3 of Figure 2.5.1

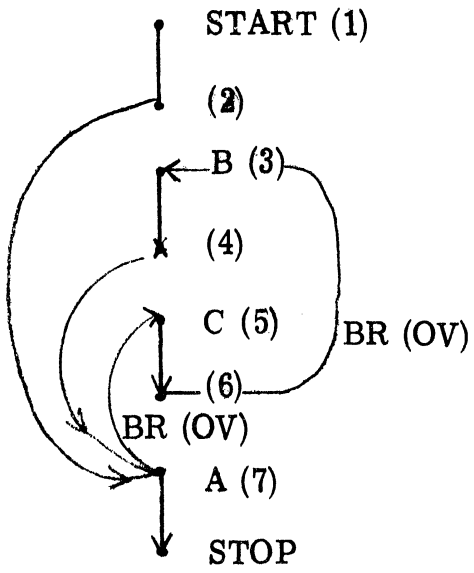
Figure 2.5.2

Program 2

| Statement # | | |
|---|---|---|
| 1 | START | S1 ◁ S2 + S4 |
| 2 | | S1 ◁ S1 SH S3, BR A |
| 3 | B | S1 ◁ −S2 + S3 |
| 4 | | S1 ◁ — S1 SH S3, BR A |
| 5 | C | S1 ◁ — S2 + S5 |
| 6 | | S1 ◁ — S1 SH S4, BR (OV) B |
| 7 | A | S2 ◁ — S1 SH S4, BR (OV) C |
| 8 | | STOP |

For this program, every add operation must be checked to determine if the add overflow condition set must be modified.

(b)




Function Performed by Blocks 2 and 3 of Figure 2.5.1 (Cont.)

Figure 2.5.2

The only statements which modify condition sets are primary predecessor statments. A predecessor statement of a condition branch is a statement which is executed before the conditional branch and whose execution could affect one of the condition sets tested by the conditional branch. A primary predecessor statement is a statement which is a predecessor statement of a conditional branch and for which there is a path from the statement to the conditional branch which contains no other predecessor statement for the same conditional branch. Thus, if there were no labels (entry points) between a conditional branch and its predecessor statements there can be only one primary conditional branch. It is true that one primary predecessor may be primary for several condition sets tested by a conditional branch. Only primary predecessor statements need to modify the state of conditional sets.

Besides branch statements which are conditional upon the state of a condition set, PTL allows branch statements which are conditioned upon a named data quantity being less than, greater than, or equal to zero. In some cases, those branch tests can be transformed into tests on the state of condition sets. In such cases, the operations performed by blocks 2 and 3 can be applied to those branch tests also.

It is difficult to estimate the amount of computer time required to execute the operations performed by blocks 2 and 3. The operations

performed are complex. However, the Fortran H compiler (L5) performs similar operations in the sense of producing flow diagrams and searching for predecessors. The definition of predecessors by Fortran H is completely different, although the search procedure would be somewhat similar. Since the compilation time of the Fortran H compiler is not excessive, we expect that the operations specified by blocks 2 and 3 are feasible in the sense of computation time.

How effective would blocks 2 and 3 be at eliminating unnecessary overhead resulting from condition set status modification? The process should be very effective. Most conditional branches which test condition sets occur within a very few statements of the operation which is being tested via the condition set mechanism. Thus, it will rarely be the case that a branch can be made to a statement between the operation being tested (via condition sets) and the conditional branch which specifies the test. Thus, in the typical case there will be only one primary predecessor (the operation statement) and the condition set overhead is minimal.

There can be more than one primary predecessor for a condition set of a conditional branch statement. If all the primary predecessors can actually be tested, no additional overhead is generated. The situation depicted in Figure 2.3.3b is an example for which there is more than one primary predecessor for a conditional branch.

There is no way to determine from the flow graph of the program whether all the primary predecessors are actually to be tested. As was pointed out earlier, such areas should be in the minority.

Thus, we feel that the operations specified by blocks 2 and 3 can be implemented with a reasonable amount of computer time. We also feel that the use of blocks 2 and 3 will eliminate most of the unnecessary overhead associated with the use of condition sets.

The fourth block represents the code insertion part of the translator. In this section each PTL operation is replaced by an in-line section of microcode. The microcode is designed to operate on fast storage, either registers or high speed local storage. Therefore, the microcode is in a symbolic form with the operands represented by their symbolic names. If there are array or string references, and calculations on the dope vector are required, the microcodes to perform those calculations are also inserted. The output from block 4 would be the microprogram, if all data were located in fast storage and were accessible by its symbolic name.

The fifth block represents the data assignment and data optimization section of the program. Each section of in-line microcode which implements a particular PTL function was designed to work upon data held in fast storage. Therefore, if the operands required

by the microcode are not contained within fast storage, two things occur. First, microcode must be inserted to move the operands from main storage into the fast storage. Second, the total execution time of the operation will be increased since time must be spent to read operands into and possibly out of main storage.

The purpose of the fifth block of Figure 2.3.1 is to assign named data quantities to storage such that the execution time of the program will be minimized. Both microtranslators and traditional compilors must perform register assignment. However, the problems of register assignment are somewhat magnified in a microtranslator for two reasons.

First, use of microcode magnifies any loss in execution speed resulting from an inefficient data assignment. The difference in execution speed for a register to register instruction versus a register storage instruction on a machine might be 3 to 1. However, the difference in speed between performing an operation in microcode with the operands in fast storage or main storage can be 10 to 1 or even larger.

Second, the assignment of named data quantities by a translator whose target language is a microlanguage will have to be more of a global assignment than that required by a traditional compiler. The local storage space available to a traditional compiler is usually

limited in size. For example, on the IBM 360 computers, there are 16 general purpose registers. Since the assignment space is limited, local assignment algorithms tend to be reasonably successful. Most local storage areas for microprogrammable computers contain 64 words or more. The larger the fast storage area the more chance that a global assignment scheme will be significantly better than a local assignment scheme.

Microprogrammable computers appear to have three levels of storage. There are the registers which are connected to the arithmetic logic units and the local storage areas which can be read into and out of the registers in a microcycle. The third storage area is main core. Thus, the assignment problem is somewhat complicated because there are two high speed storage areas (registers and local storage). However, the number of registers are usually limited and must be assigned locally and not globally.

The fifth block has special significance with respect to the integration of subroutines with a main program. It would be very nice if the assignment of named data quantities could be done globally, even across subroutine linkages. This can be accomplished if the assignment of named data quantities is done for both the main program and its subroutines at the same time. Thus, the fifth block

starts to take on the characteristics of a very sophisticated loader. The procedure would work as follows. Subroutines would be translated up to the beginning of block 5 and stored until the main program is translated. Then a global assignment is accomplished by combining the main program and its subroutines as entries into the fifth block of the translator.

There is reason to question the necessity of a "main program and subroutines named data quantity assignment" procedure. The importance of the procedure will depend upon five factors: the number of operands within the subroutine; the amount of high speed local storage; the location of subroutines relative to loops; the frequency of execution of the subroutines; the number of independent calls of each subroutine.

The number of operands within the subroutine and the amount of local storage determine the usefulness of a global assignment scheme. If the number of operands and the space required for their storage is larger than the amount of local storage, the use of global assignment would be nearly equivalent to an assignment scheme local to the subroutine. If the number of operands and the space required for their storage is small and the amount of local storage is large, the use of global assignment across subroutine

boundaries would be beneficial.

If a subroutine is located within a tight loop or if a subroutine has a high frequency of execution the effect of data assignment for the subroutine will be more pronounced.

The number of independent calls of each subroutine affects the difficulty of a global assignment of named data quantities. The operands of a subroutine are bound. Therefore, if a global assignment is attempted the procedure must attempt to work out starting with the subroutines and then assigning operands for each of the calling areas. Such assignments would be tricky at best.

Is it possible to develop a sophisticated global data assignment procedure? The answer is yes. Several register assignment schemes have already been implemented. The IBM Fortran H compiler attempts to optimize register assignment. There are only 16 general purpose registers and 4 floating point registers so the optimization must be somewhat localized in the Fortran H compiler. There is a Fortran compiler which optimizes register assignment for the Univac 1108. The 1108 has 48 registers, so the assignment must be somewhat global. It is interesting to note that the compiler can optimize for the main program and its subroutines as a unit. There are several register assignment procedures [ D4] which have

been proposed by Day. The speed of the algorithms developed by Day is heavily dependent upon the profit vector which associates a profit for the assignment of an operand to a register. Therefore, it is difficult to predict the value of his assignment algorithms. However, he has established a theoretical base from which to proceed. Thus, there are in existence one or more compilers, which perform global assignment and there is a theoretical base from which new global assignment algorithms may be developed.

The discussion up to now has assumed that PTL programs would be translated into microcode. However, the term microprogramming and microcode refer to two different types of microprogramming and microcode. The two types of microprogramming are vertical microprogramming and horizontal microprogramming. A microprogram system is seldom either vertical or horizontal, but a type lying between the two. (See Appendix C.)

The Program Type Language (PTL) can be translated into either vertical or horizontal microcode. However, the translation into one or the other will affect both the translation process and the resultant microcode. We wish to compare and contrast some of the differences between the translation of PTL programs into vertical microprogramming and horizontal microprogramming.

Horizontal microprogramming allows more complete control of the CPU than vertical microprogramming. In particular, parallelism can be exploited more easily by using horizontal microprogramming rather than vertical microprogramming. Thus, the coding of the in-line section of microcode for each program type can on the average be done more efficiently using horizontal microprogramming. If horizontal microprogramming is available, the translation program should attempt to take advantage of parallelism when the in-line sections of microcode (program types) are merged to implement a program. The translator needs to test the last microinstruction for the current in-line section of microcode and the first microinstruction for the next section of in-line microcode to determine if parallelism would allow the two instructions to be merged into one microinstruction. Any conditional branching associated with the two microinstructions must also be checked for compatability.

Both horizontal and vertical microprogramming allow access to high speed local storage. Thus, for both types of microprogramming operand assignment is very important. For horizontal microprogramming, it may require an equivalent of 10 microcycles to transfer an operand from main core into high speed local storage.

Therefore, the difference in execution speed between maintaining an operand in high speed local storage or main core may be very large. For vertical microprogramming, it may require an equivalent of only 1 microcycle to transfer an operand from main core into high speed local storage. This situation occurs when each vertical microinstruction resides in main core. Even for vertical microprograms which reside in main core, the assignment of operands can make a significant difference in the overall execution speed.

Both vertical and horizontal microprogramming, when used to implement PTL programs, eliminate one level of interpretation. Thus, PTL program types are implemented directly in microcode, instead of being implemented in machine language instructions which would then be implemented in microcode. If the vertical microinstructions are really miniinstructions then less than a full level of interpretation is eliminated. The miniinstructions provide finer control of the CPU than machine instructions but they do not operate at the gate level as do horizontal microinstructions.

From the previous discussion, it is apparent that the translation of PTL programs into horizontal microcode will produce faster executing programs than vertical microcode for equivalent machines. However, the length of each control memory word for horizontal microcode is

two or three times greater than control memory for vertical micro-programming. Thus, the cost of control memory for large programs is somewhat prohibitive if horizontal microprogramming is utilized.

# Chapter III

# A MODEL OF MICROPROGRAM CONTROL

One of the objectives of this study is the optimal selection of components for central processing units. Therefore, a cost performance model of the central processing unit must be devised. We will view the CPU as a collection of hardware units which are controlled by a microprogram. The functions to be performed by the CPU will consist of a set of program types. Each program type (Chapter II) is expected to be a frequently used computer operation.

The CPU hardware configuration will be modeled as a base hardware unit and a set of optional hardware units. Each program type will have one or more implementations. A program type implementation is a sequence of microinstructions which will cause the CPU hardware to perform the desired function. Each of the implementations for a particular program type will use different hardware options and will have a different execution time and a different hardware cost. The execution time and cost of a program type implementation can be computed and the optimal cost performance curve for a set of program types can be computed.

The preceding paragraphs have been a brief description of what will be developed in this chapter. The development will proceed as

follows. First, the general model will be presented. Second, the appropriate definitions will be given. Third, the equations of cost and execution time will be developed. Fourth, the problems associated with the selection of a base hardware set will be discussed.

## 3.1 The Model

The model is composed of two parts: the operations model and the CPU model. The operations model describes the basic functions to be performed by the CPU. The CPU model describes the available hardware with which operations may be implemented.

The operations model is the Program Type Language (PTL) with the addition of statistics on usage. Each PTL operation is one function to be performed by the CPU. Since any non-PTL operation is described using PTL operations and hence is executed using PTL operations, PTL operations are constrained to represent the basic functions performed by the CPU. This emphasizes the importance of selecting PTL operations wisely. The statistics on usage are required to calculate performance and are described by values for both frequency of occurrence and frequency of execution. These quantities will be defined formally in Section 3.2. The information supplied by the statistics on usage allows the performance calculations to weigh each program type as a function of its frequency of execution. The statistics on usage are also used in the storage cost calculation since they contain information concerning the number of occurrences of each program type for a general class of microprograms.

The CPU model is used to describe the CPU or rather a collection of possible CPU's. The CPU is modeled as a base hardware

and a set of possible hardware options. The hardware options are functional units which may be added to the CPU to improve the performance. The base hardware may be as primitive or as sophisticated as desired. However, it is generally true that a more primitive base hardware will induce more hardware options than would a sophisticated base hardware.

Each hardware option has a cost. The hardware option cost is composed of two parts. First, there is the actual cost to physically construct the unit and add the unit to the base hardware. Second, there is the additional cost of control memory resulting from any additional microorders required by the hardware option. The additional microorders result in a lengthening of the microword. The CPU is modeled by specifying a base set of hardware, the cost of the base hardware, and the number of microorders required to control the base hardware. Also specified are a set of hardware options, the cost of each hardware option, and the number of microorders required to control each hardware option.

The two parts of the model are connected by the implementation methods. The operation model specifies what functions are to be performed, how often each function occurs in a program, and how often each function is executed. The CPU model specifies the available hardware and the cost of the hardware.

Each implementation method specifies the microinstructions required to implement a particular operation using some subset of the hardware. Therefore, an associated cost in terms of control storage cost and hardware costs and an execution time in terms of microcycles can be computed for each implementation.

In the next section formal definitions will be given for the terms discussed above. Also, the cost equations will be given in the same section.

## 3.2 Cost Performance Equations

In this section, the cost performance equations will be developed. The equations represent the cost in dollars versus the execution time in microcycles for the implementation of a specified set of program type operations. Before developing the cost performance equations, a set of definitions will be given.

The first group of definitions involve program type implementations. The program type vector P is a list of all program types. The hardware option vector H is a list of all available hardware options. Each hardware option is a piece of circuitry which may be added to the basic CPU configuration.

The designers will supply one or more implementation methods for each program type $P_j$. Each implementation method is a sequence of microinstructions which will cause the CPU to perform the desired function. It is required that the designers specify for each program type one implementation method that uses only the base hardware. The hardware designer may then specify "improved" sequences of microinstructions that require one or more of the available hardware options. The improved sequences are expected to either use fewer control memory words or execute faster than the basic implementation method or both.

The specification of the hardware options required for each implementation method is given by the implementation matrix $B^j$. One row of each implementation matrix $B^j$ will contain all zeroes since one implementation method uses only the base CPU configuration.

Program Type; P: A program type vector P is the list of statements allowed in the algorithm language.

$P_j$ = the jth program type.

Hardware Option; H: The hardware option vector H is the list of available hardware options.

$H_j$ = the jth hardware option.

Implementation Matrix; $B^j$: The implementation matrix $B^j$ specifies which hardware options are required for the implementation methods for the jth program type. The ith row of matrix $B^j$ specifies which hardware options are required for the ith implementation method of the jth program type.

$B^j_{\ell k}$ = 1 the kth hardware option is required for the $\ell$th

implementation method of the jth program type.

$B_{\ell k}^{j} = 0$ the kth hardware option is not required for the $\ell$th implementation method of the jth program type.

Extensive use of subsets of the hardware option set H will be made throughout the remainder of this work. The hardware option select vector $R^i$ provides a convenient way of specifying such a subset. The superscript i is used to order any set of select vectors and is used for identification purposes only. Even though $R^i$ is a vector it will be convenient to occasionally refer to the set of hardware options specified by $R^i$ as the set $R^i$.

H Option Select; $R^i$ : The H option select vector $R^i$ specifies a choice of hardware options.

$R_k^j = 1$, the kth hardware option ($H_k$) is selected.

$R_k^j = 0$, the kth hardware option ($H_k$) is not selected.

The superscript i is used to order any set of select vectors.

The cost of the hardware is the cost of the base hardware plus the cost of any hardware options (denote that subset of H by $R^k$)

which are chosen to complement the base hardware. The cost of an individual hardware option may be dependent upon the other elements of $R^k$. For example, there may be two hardware options which can be combined into one package if both are chosen. Thus, the cost of both hardware options would be less than the sum of their individual costs by the amount of the reduction in packaging costs.

The ability to express dependent cost relationships among the hardware options is achieved by providing a vector of costs for each hardware option. Each element of the cost vector $C^i$ of the ith hardware option has two associated vectors. The inclusion vector $U^i_j$ specifies which hardware options must be present in $R^k$ if the jth element of $C^i(C^i_j)$ is to be used as the cost of the ith hardware option, and the exclusion vector $X^i_j$ specifies which hardware options must not be present in $R^k$ if the jth element of $C^i(C^i_j)$ is to be used as the cost of the ith hardware option. The set of inclusion vectors and exclusion vectors required by $C^i$ are denoted by $U^i$ and $X^i$ respectively. Both $U^i$ and $X^i$ are also used with the microbitivector $M^i$ which is defined later.

The hardware option cost is given by Equation 3.2.1 which follows the definitions of $C^i, U^i, X^i$, and $\rho(R^k, U^i_j, X^i_j)$ given below.

Hardware Option Cost; $C^j$: The hardware option cost vector $C^j$ lists the set of possible costs for the jth hardware option.

The Inclusion Matrix; $U^j$: The ith row of the inclusion matrix $U^j$ specifies which hardware options must be included with the jth hardware option if the ith cost and the ith microbit value for the jth hardware option are to be used.

$U_{ik}^j = 1$ the kth hardware option must be included with the jth hardware option if the $C_i^j$ cost value and the $M_i^j$ microbit value are to be used.

$U_{ik}^j = 0$ the kth hardware option may or may not be included with the jth hardware option if the $C_i^j$ cost value and the $M_i^j$ microbit value are to be used.

The Exclusion Matrix; $X^j$: The ith row of the exclusion matrix $X^j$ specifies which hardware options must not be included with the jth hardware

option if the ith cost value and the ith microbit value for the jth hardware option are to be used.

$X_{ik}^{j} = 1$ the kth hardware option must not be included with the jth hardware option if the $C_i^j$ cost value and the $M_i^j$ microbit value are to be used.

$X_{ik}^{j} = 0$ the kth hardware option may or may not be included with the jth hardware option if the $C_i^j$ cost value and the $M_i^j$ microbit value are to be used.

The $\rho$ function $\rho(R^k, U_j^i, X_j^i)$ has a value of 1 if the set of hardware options specified by $R^k$ satisfies the requirements of hardware option inclusion and exclusion specified by $U_j^i$ and $X_j^i$ and is 0 otherwise. The $\rho$ function is used to select the cost of a hardware option (i) from its cost vector ($C^i$) and the required microbits from its microbit vector ($M^i$ to be defined later).. It is natural to expect that $\rho(R^k, U_j^i, X_j^i)$ will be zero if the ith hardware option is not specified ($R_i^k = 0$). This can be accomplished by requiring that $U_{ji}^{i} = 1$ for all j. Also, the set $U^i$ and the set $X^i$ must satisfy the condition

that there is at most one value of j for each possible $R^k$ such that $\rho(R^k, U^i_j, X^i_j)$ is 1. For example, if there were four hardware options and the cost of hardware option 3 depended upon the inclusion or exclusion of hardware options 1 and 2 but was independent of hardware option 4, the $U^3$ and $X^3$ matrices might appear as below.

$$U^3 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \qquad X^3 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The cost value $C^3_1$ would be selected if $R^k$ were (0010) or (0011), $C^3_2$ would be selected if $R^k$ were (0110) or (0111), etc.

$\rho$ function; $\rho(R^\ell, U^i_j, X^i_j)$: The $\rho$ function gives a value of 1 if $R^\ell$ satisfies the requirement specified by $U^i_j$, and $X^i_j$.

$$\rho(R^\ell, U^i_j, X^i_j) = 1 \text{ if } \sum_k (\overline{R^\ell_{jk}} \cdot U^i_{jk}) = 0$$

and

$$\sum_k (R^\ell_{jk} \cdot X^i_{jk}) = 0$$

otherwise = 0

3.2.1 Cost of hardware option i for option set $R^k$ $= \sum_j c_j^i \cdot \rho(R^k, U_j^i, X_j^i)$

Note that $\rho(R^k, U_j^i, X_j^i)$ will be 1 for only one value of j.

Occurrence Vector; O: The occurrence vector O is the estimated number of occurrences of each program type in a typical program.

$O_j$ = the number of occurrences of the $P_j$ in a "typical program".

Execution Frequency Vector; F: The execution frequency vector F is the estimate of the percentage of program type executions which will be spent in each occurrence of a program type.

$F_j$ = estimate of the percentage of program type executions which will be spent in each occurrence of the jth program type.

Execution Time Vector; $V^j$: The execution time vector is the estimated number of microcycles required to perform the jth program type function

for each implementation method.

$V_i^j$ = the estimated number of micro-cycles required to perform program type j using the ith implementation method.

The cost of the CPU includes the base hardware cost ($c_b$), the cost of hardware options, if used, and the cost of the control memory. The cost of the control memory is a function of the cost of each bit of control memory ($c_s$), the length of the microprogram, and the length of the microinstruction word.

The microinstruction word length is determined by the number of bits required by the base hardware and the number of bits required by the hardware option set ($R^k$). The base hardware bits are specified by $m_b$ and include the microorders required to control the base hardware and to sequence the microprograms. The number of microbits required by an individual hardware option may be dependent upon the other elements of $R^k$. For example, there could be one hardware option (the addition of a data path bus) which would force the addition of data path selection circuitry in other hardware options which would require an additional microorder.

The ability to express dependent microbit relationships among the hardware options is achieved by providing a vector ($M^i$) of microbit values for each hardware option. The inclusion matrix ($U^i$), the exclusion matrix ($X^i$), and the $\rho$ function are used to determine which element of $M^i$ to use as the microbit value. The microbits for the ith hardware option are given by Equation 3.2.3. The concepts employed are identical to those used to select the cost value for the ith hardware option.

Hardware Option Microbits; $M^j$: The hardware option microbit vector $M^j$ lists the set of possible microbits for the jth hardware option.

3.2.2 Microbits for option i for option set $R^k$ $\quad \sum_j M^i_j \cdot \rho\,(R^k, U^i_j, X^i_j)$

In specifying the microinstruction word length it is possible to include both field encoding and word encoding (Appendix C). Field encoding is handled by allowing the hardware option designers to specify the number of microbits required for each hardware option in place of the number of microorders. Thus the designers can field encode the information required to control a hardware option in any manner that they desire. Normally a limit, k, is set on the maximum number of bits per field. This requirement results from speed considerations; the more bits per field, the more gate

delays in the decoding process

The word encoding is handled by allowing the designer to specify a value for w, which is called the word form variable. If the value of w is greater than 1, there is more than one word type. If there is more than one word type, a vector $W^j$ of 1's and 0's for each hardware option (j) is supplied. The $W^j$ vector specifies to which (there may be more than 1) of the w word forms the microbits required by the jth hardware option should be added. Thus if w were 2, and the base microbits were 10, and hardware option 1 added 3 bits to word form 1, and option 2 added 2 bits to word form 2, the length of the microword for the base hardware plus options 1 and 2 would be 13 bits. However, if the microbits required by hardware option 1 were to be added to both word forms 1 and 2, the length of the microinstruction for the base hardware plus option 1 and 2 would be 15 bits.

Word Form; w: The word form, w, specifies the number of different word types used by the base hardware.

Word Type; $W^j$: The work type, $W^j$, specifies to which of the word types the microbits required by the j hardware option are to be added. $W^j_i = 1$ add the microbits of the jth hardware option to word type i.

$W_i^j = 0$ do not add the microbits of the jth hardware option to word type i.

Base Hardware Costs; $c_b$: The base hardware cost $c_b$ is the cost of the basic CPU hardware configuration including the hardware required for the basic microsequencing.

Control Memory Bit Costs; $c_s$: The control memory bit cost $c_s$ is the cost per binary storage element of the control memory storage array.

Base Microbits; $m_b$: The number of microbits required to control the base set of hardware and to sequence the microprograms.

Field Encoding Limit; k: The field encoding limit k is the maximum number of bits per field.

The concept of microsubroutine needs to be explored briefly. The microprogram implementation of individual PTL operations required by a program written in the program type language can be accomplished in two ways. Each operation can be replaced by an in-line section of microcode or each operation can be replaced by linkages to the proper microsubroutine. One or the other must be

done for each occurrence of an operation within the program.

The usage of microsubroutines results from a desire to reduce the control storage requirements. If only one copy of the microinstructions required to implement each operation is maintained in the control store, and if the number of microinstructions required to branch to a subroutine is small, a major reduction in control storage requirements is possible.

The replacement of in-line microcode by microsubroutines will cause a decrease in the execution speed of the program. The decline in execution speed results from two major factors. First, each linkage to or from a microsubroutine requires one or more microcycles. Second, the use of microsubroutines makes it a more difficult task to optimally allocate high speed memory (register and scratch pad memory) for either temporary or permanent storage of data. Therefore, additional microcyles may be required to move data from core storage to high speed memory for processing if sufficient high speed storage is not available. The two points mentioned above will be discussed in the next few paragraphs.

If microsubroutines are used to implement a PTL operation, each occurrence of the operation within a program will be replaced by a branch to the microsubroutine. The microsubroutine will effect the performance of the required operation and a return branch to

the next PTL statement is then executed. Thus, an additional overhead of subroutine linkage occurs when microsubroutines are used in place of in-line microcode. It is possible to add sophisticated microroutine branching hardware to reduce the number of microcycles required to branch to and from a microsubroutine. However, the addition of hardware offsets the saving in control storage resulting from the use of microsubroutines.

If a microroutine is to be efficient in terms of performance any data required by the microroutine should be located in the highest level of storage. Microroutines are written under the assumption that data is in either high speed registers or scratch pad memory. If the microroutine is written to be used as in-line code, the exact specifications of storage locations for data can be accomplished when the PTL program is compiled into microcode. In fact, an important part of the compilation process would be the optimal allocation of high speed storage registers and scratch pad memory to achieve efficient execution.

However, if the microroutine is written as a microsubroutine, the location of the data must be fixed without regard to the intended PTL program. It would be difficult, if not impossible, to make efficient use of high speed storage if in-line microcode is replaced by microsubroutines. Therefore, the average execution time of

PTL operations might deteriorate because of excessive references to main core. Nevertheless, the savings in control storage that result from microsubroutines might be significant, or in certain cases, the reduction in performance might not be significant.

We sum up with the following observations. Many of the PTL operations are easily implemented and would require very few lines of microcode. For these operations it would seem that implementation should be in the form of in-line microcode. One of the costs associated with microcode, in terms of control storage and execution time, is the need to move data into and out of the high speed storage areas. Since microroutines are written to work on data already contained in high speed storage, the storage costs and the execution time attributed to micro-routines do not reflect the cost of data movement between main core and high speed storage. It should be remembered that a PTL program compiled into in-line microcode can optimize the location of data so as to minimize data movement between main core and high speed storage.

Some operations may be so complicated and occur so often in a program that very large amounts of control storage can be saved. If these operations are few in number it may be the case that a slight degradation in performance is acceptable, or it may be

possible to improve the performance by reserving some of the high speed storage for the exclusive use of microsubroutines. In general, we believe that the use of microsubroutines should be avoided if possible.

The use of microsubroutines to implement program type operations should not be confused with PTL subprograms. A PTL subroutine would be composed of many program type operations any of which could be implemented as a microsubroutine. Thus, microsubroutines are a method of implementing program type operations, not PTL subroutines.

The decision to use in-line microcode or microsubroutines to implement each occurrence of a program type is made a priori for each program type. The choice is indicated by the subroutine selection vector.

Subroutine Selection Vector; E:

The subroutine selection vector E indicates which program types will be implemented as in-line code and which program types will be implemented as subroutines.

$E_j = 1$ The jth program type will be implemented as microsubroutines.

$E_j = 0$ The jth program type will be implemented as in-line sections of microcode.

The calculations involving microsubroutines require the definitions of several new variables. The definitions require no elaborations and are listed below.

Subroutine Branch; $s_B$: The subroutine branch $s_B$ gives the number of microinstructions required to initiate a subroutine branch.

Subroutine Return; $s_R$: The subroutine return $s_R$ gives the number of microinstructions required to initiate a return from a microsubroutine to the calling area.

The other value needed for calculating the amount of storage is the storage vector $S^j$. The storage vector gives the number of microinstructions required for each implementation method of the jth program type.

Storage Vector; $S^j$: The storage vector $S^j$ lists the number of microinstructions required for each implementation of the jth program type.

$S_i^j$ = the number of microinstructions required for the ith implementation method of the jth program type.

The last definition is that of the selection vector. The selection vector is used to simplify the equations of cost and performance. It lists the current choice of program type implementation.

Selection Vector; $N^i$: The selection vector $N^i$ lists the current choice of the implementation method for each program type.

$N_j^i$ = current choice of the implementation method for the jth program type.

The superscript i is simply used to order any set of selection vectors.

The cost performance equations will be developed for two cases. The difference between the two formulations is the use of special branching hardware. The effect of special branching on cost is a decrease in the length of control storage, and hence, cost and the effect on performance is usually an increase in average execution time. The cost equations will be developed for both subroutine branching and no subroutine branching. Then the execution time equations will be developed. The selection vector is used exten-

sively since the cost equations and the average execution time equations are being developed for a particular set of implementation methods. However, the equations are valid for all combinations of implementation methods.

Equation 3. 2. 3 calculates the hardware options ($R^{\ell}$) required by the set of implementations specified by the selection vector ($N^{\ell}$). The cost(CH($R^{\ell}$)) of the base hardware and the hardware options is given by Equation 3. 2. 4. The cost of the hardware options is calculated by summing the cost (Equation 3. 2. 1) of each hardware option specified by $R^{\ell}$. Remember that $R^{\ell}$ is a vector with a 1 in position j if the jth hardware option is a member of the current set of hardware options.

Equations 3. 2. 5 and 3. 2. 6 are used to calculate the width (number of bits) of the microinstruction word. If word form (w) is a 1, the number of bits is found by summing the base microbits ($m_b$) and the microbits (Equation 3. 2. 2) required by the hardware options specified by $R^{\ell}$.

If word form is greater than 1, the width of each word is computed, and the number of bits of the largest word is the microinstruction word size. The width of the jth word (with $w > 1$) is found by summing the base microbits and the microbits required by the hardware options specified by $R^{\ell}$ and assigned to the word by $W_j^i$.

The cost of control memory for the two cases is given by Equations 3.2.7 and 3.2.8, respectively. If microsubroutines are not used ($CNS(N^{\ell})$ Equation 3.2.7) then the number of occurrences of each program type and the number of microinstructions for the implementation method for each program type are used to calculate the length of the program. If microsubroutines are used ($CSS(N^{\ell})$ Equation 2.2.8), then the number of occurrences of each program type and the number of microinstructions required for each subroutine call, plus the number of microinstructions for each program type implementation are used to calculate the length of the program.

The execution time ($TNS(N^{\ell})$) of the jth program type as a function of $N^{\ell}$ is given by Equation 3.2.9 for nonmicrosubroutine implementations. When subroutine branching is used the value given by Equation 3.2.9 must be increased by the number of cycles required to initiate a subroutine call plus the number of cycles required to initiate a subroutine return. The number of cycles to branch is assumed equal to the number of microinstructions to branch ($s_b$) and the number of cycles to return is assumed equal to the number of microinstructions to return. This is reasonable if both the branch code and the return code were straight line code. That is given by Equation 3.2.10.

The value $\overline{T}$ (Equation 3.2.11) is the average number of micro-cycles per program type execution. The estimate is made by using the frequency of occurrence, frequency of execution, and the estimated execution time of each program type. The sum over all program types of the product of those three quantities is divided by the total number of program type executions to arrive at the average execution time (Equation 3.2.11). The performance measure PT is the reciprocal of the average execution time.

3.2.3 $$R_k^{\ell} = \bigvee_{j=1}^{|P|} B_{N_j^{\ell} k}^{j}$$

$|P|$ = length of program type vector

3.2.4 $$CH(R^{\ell})_{\text{Hardware}} = c_b + \sum_{j=1}^{|H|} \left( \sum_k C_k^j * \rho(U_k^j, X_k^j, R^{\ell}) \right)$$

if $w = 1$

3.2.5 $$M(R^{\ell}) = m_b + \sum_{j=1}^{|H|} \left( \sum_k M_k^j * \rho(U_k^j, X_k^j, R^{\ell}) \right)$$

if $w > 1$

3.2.6 $$M(R^{\ell}) = \max_{\text{over } i} \left( m_b + \sum_{j=1}^{|H|} \left( \sum_k M_k^j * \rho(U_k^j, X_k^j, R^{\ell}) * W_i^j \right) \right)$$

3.2.7 $$CNS(N^{\ell})_{\substack{\text{Storage} \\ \text{Non-Sub}}} = \left( \sum_{j=1}^{|P|} ((O_j * S_{N_j^{\ell}}^j) \wedge \overline{E}_j) \right) * M(R^{\ell}) * c_s$$

3.2.8 $$CSS(N^{\ell})_{\substack{\text{Storage} \\ \text{Subroutine}}} = \left( \sum_{j=1}^{|P|} ((O_j * S_B) + S_{N_j^{\ell}}^j + S_R) \wedge E_j) \right) * M(R^{\ell}) * c_s$$

3.2.9 $$TNS(N_j^{\ell}) = V_{N_j^{\ell}}^{j}$$

3.2.10 $$TS(N_j^{\ell}) = V_{N_j^{\ell}}^{j} + s_B + s_R$$

$$3.2.11 \quad \overline{T}\ (N^{\ell}) = \frac{\sum_{j=1}^{|P|} (F_j \cdot O_j[(V^j_{N^{\ell}_j} \wedge \bar{E}_j) + (V^j_{N^{\ell}_j} + s_B + s_R) E_j]}{\sum_{j=1}^{|P|} F_j \cdot O_j}$$

$$3.2.12 \quad PT = \frac{1}{\overline{T}} \quad \text{performance measure}$$

## 3.3 Base Hardware

In the previous section the existence of a suitable base hardware set has been assumed. In this section the base hardware concept will be examined further. In particular, two points will be discussed. First, what would constitute a minimal base set? Second, what procedures would have to be invoked if one or more program types could not be implemented using the base hardware set?

A base hardware set is by definition a set of hardware sufficient to allow the implementation of any program type operation. Therefore, a minimum base hardware set is a hardware set which satisfies the completeness criteria above and, in addition, minimizes some other function. We assume that the other function is cost. If one is strictly interested in the minimum cost set of base hardware, it might be possible to examine and compute the cost of all possible minimum base hardware sets. However, such a path seems very extreme and not terribly revealing. Instead, a near minimum set of base hardware will be presented and examined.

The base hardware set is shown in Figure 3.3.1. The hardware consists of the core memory, three registers, the arithmetic logic unit, the data paths, the microprogram controller, and the

microprogram storage. Three registers are the minimum number of registers required to access core memory and also perform logical operations upon core memory data. The three registers are memory address register, a memory buffer register, and a result register. The MB and MA registers are required to cycle the core memory. The results register is needed to provide two inputs into the ALU. The MA register cannot be used to provide the second data input into the ALU, since once data was loaded from the MB into the MA, the second data word could not be loaded into the MB, since there would be data in the memory address register (MA), not the address of the second data word. The use of only three registers implies that the microprograms will use certain fixed locations in core as temporary locations, program counter, etc. These locations will be accessed by loading the MA using preset valves supplied by the emit fields. The emit field is a field within the microprogram instructions which allows the programmer to provide fixed (immediate) data for program use. It should be mentioned that there are hidden registers required by the microprogram control unit. A minimum of two registers would be required; one to hold the control information and one to hold the address of the next control storage word.

The data paths are bus connections into and out of the ALU. The emit data path allows the microprogram to load immediate data from the control storage through the ALU into any register. There would seem to be little room to simplify the data paths. The cost of the base hardware might be reduced by providing for serial transfers instead of parallel transfers of data.

The ALU is assumed to have the ability to perform logical AND, as well as to be able to perform the complement operation. In addition, the output of any bit of the ALU may be tested and branched upon by the microprogram. It is assumed that only one bit of the result may be tested at a time. With the hardware described above a shift would be performed by loading the RR register with all ones (from the emit field), loading the word to be shifted into the MB register, testing each bit of the MB (by passing the MB through the ALU), and then after each test, performing a logical AND between RR and the proper emit word from control storage and storing the result in RR. The final step would be to transfer the RR register to the MB register. An example of a program to do a 1 bit left shift is given in Table 3.3.1. Programming using the base hardware set of Figure 3.3.1 would be tedious to say the least. The microprogram to implement the shift operation would be lengthy and would include many locations of emit (immediate) data. Thus, the savings in base hardware would be partially absorbed by an increase in control storage.

The testing which is available allows the microprogram to perform a branch conditioned upon the status of any bit of the ALU output. However, only one bit of the ALU output may be tested at a time. If a shift capability were provided with the ALU, only one bit of the ALU would need to be tested, since the other bits could be tested by shifting the desired bit to the test position. Such a test setup would increase the size of the microprograms.

The ALU could be replaced by a table look up facility. The table location could be specified by a field in the microinstruction and the contents of the MB and RR registers. Such a setup might reduce the base cost, but again the base cost is being reduced by increasing the storage requirements since added storage would be required to hold the look up tables.

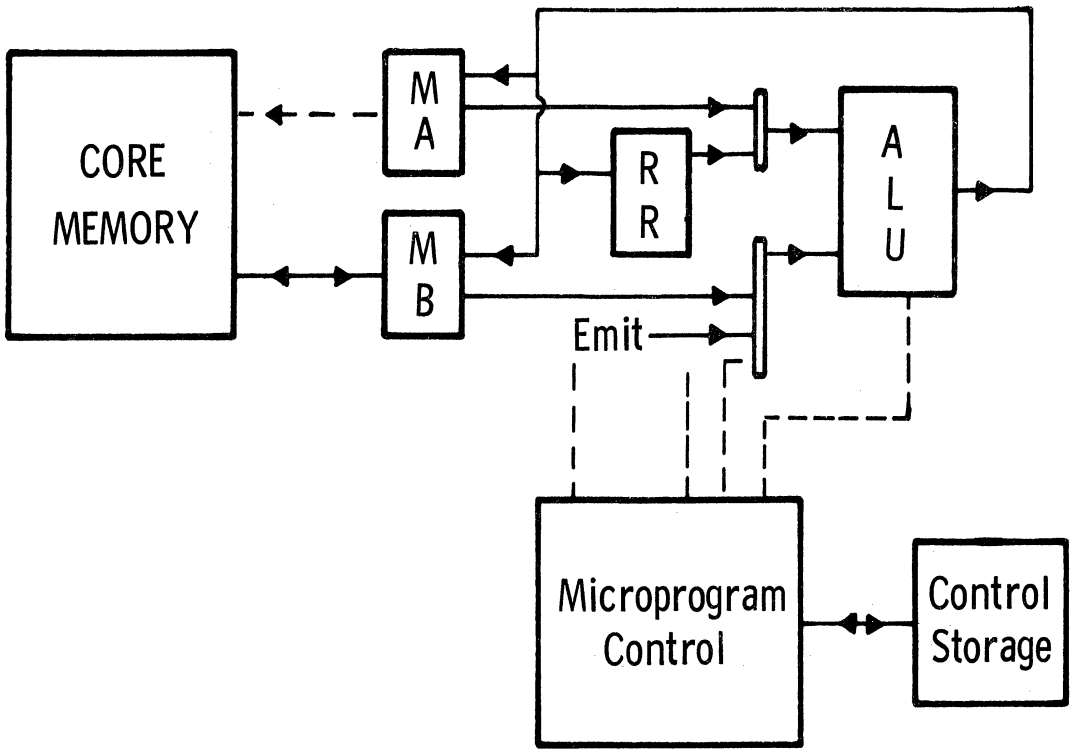
The ALU could also use different sets of basic operations. The ALU might implement logical OR, as well as complement. So the number of possibilities for a "minimal" base hardware set are rather extensive.

Now let's consider the question of an insufficient base hardware set. Such a condition could occur in two ways. First, the hardware may not have the capability to allow the implementation of one or more program type operations. That shall be called an insufficient hardware set. Second, the programmer may feel that to implement a particular operation on the base hardware is too tedious and too ridiculous

to attempt. The thought of performing a multiply on the base hardware set of Figure 3.3.1 is rather terrible to imagine.

If the base hardware set is insufficient, a change in the optimization algorithm is required. Two different problems may arise. First, some hardware option may be essential and thus should not be a hardware option but should be made part of the base hardware set. An essential hardware option is a hardware option which required by each implementation method of a program type. Such a hardware option should be part of the base hardware.

Second, if there are no essential hardware options or if the essential hardware options have been removed and there are still program types which cannot be implemented using the base hardware set, the search algorithm in the optimization program must be modified. This means that there are program types which do not have a base implementation. This would occur if the programmer felt that it was unreasonably difficult (or slow) to implement the program type on the base hardware. The modification will simply require the skipping of any set of hardware options which does not have at least one feasible implementation for each program type operation. This could be effected by adding a dummy base hardware implementation which required $10^6$ units of storage and $10^6$ microcycles to execute.




Minimum Base Hardware

Figure 3.3.1

Assume Registers are 4 bits

```
        LOAD   RR              '1111'
        IF   MB1  = 0          BR   L5
L1      IF   MB2  = 0          BR   L6
L2      IF   MB3  = 0          BR   L7
L3      IF   MB4  = 0          BR   L8
L4      TRANSFER  RR   TO   MB
        BRANCH  TO  NEXT  PROGRAM
L5      AND  RR  WITH   '1101'
        BR       TO     L1
L6      AND  RR  WITH   '1011'
        BR       TO     L2
L7      AND  RR  WITH   '0111'
        BR       TO     L3
L8      AND  RR  WITH   '1110'
        RR       TO     L4
```

LEFT SHIFT 1 END AROUND

Table 3.2.1

# Chapter IV

# OPTIMIZATION

## Introduction

The equations of cost and performance have been developed in the preceding chapter. The goal of this chapter is to develop optimization procedures capable of determining the minimum cost hardware configuration given a desired performance goal. For example, if the average execution time per program type is not to exceed 3.7 usec; the optimization procedure will select the least expensive configuration including control storage cost commensurate with the 3.7 usec execution time.

The chapter will proceed as follows. First, the general problem and the magnitude of the optimization space will be discussed. Second, the general approach to the optimization will be discussed. Third, the optimization algorithm will be presented. Fourth, a special optimization technique which is part of the overall optimization will be discussed.

## 4.1 General Problem

The optimization problem is the selection of the least expensive hardware configuration including control storage which achieves a desired performance goal. The execution time (equation 3.2.11) is the weighted average of the number of microcycles for each program type. Each program type execution time is weighted according to the frequency of execution of each program type. The hardware configuration cost includes the cost of control storage. Thus, each different implementation method for a program type can affect the hardware configuration costs in two ways. First, the hardware options, if any, required by the implementation method affect the total hardware option cost. Second, each implementation method requires a specific amount of control storage and thus affects the storage costs.

The space of possible implementations is very large. If each of 50 program types had only 2 implementation methods (a small estimate), the state space would contain on the order of $10^{15}$ points. Thus, some efficient method of searching this very large space must be developed.

Before continuing the discussion of the optimization problem the optimization variables will be discussed briefly. Figure 4.1.1 is the input to the optimization program presented in an easily understood format.

The first section lists the hardware options, the cost of each hardware option, and the number of additional microorders required by each hardware option. A separate list would explain the function or functions performed by each hardware option including an explanation of the required microbits. In the example of Figure 4.1.1 the 6th hardware option which is a floating point add/subtract unit costs 2000 units and requires three microbits which are added to the 1st and 2nd word forms. The cost of a hardware option may depend upon the inclusion of other hardware options. For example (in Figure 4.1.1) hardware option 1 will cost 300 units if hardware option 5 were not also chosen and will cost 380 units if hardware option 5 is used with hardware option 1. The microbits (1 if $\overline{5}$, 2 is 5) required by option 1 are also dependent upon option 5.

The value of $\omega$ specifies how many different word forms (word encoding is discussed in Appendix C and Section 3.2) are used by the base hardware microinstruction word format. With multiple word forms, the microbits required by a hardware option are associated with one or more word forms. In the example (Figure 4.1.1) the microbits required for hardware option 6 are added to word forms 1 and 2.

The next section specifies the storage and time required to use microsubroutines. A microsubroutine is the replacement of in-line implementation of a program type by a branch to a microsubroutine. The use of microsubroutines to implement program types results in a reduction in control storage requirements but causes a reduction in

performance. The number of instructions required to initiate a branch to a microsubroutine and the number of instructions required to initiate a return from a microsubroutine are specified. These values are used to calculate the effect of implementing a program type as a subroutine instead of replacing each occurrence of a program type by an in-line section of microcode.

The remaining sections of Figure 4.1.1 contain information relative to each program type implementation. The first row gives the program type number, its frequency of occurrence, frequency of execution, and type. The frequency of occurrence specifies the number of times the program type appears in the program. The frequency of execution specifies the average number of executions for each occurrence. The type specifies whether the program type will be implemented as an in-line section of microcode (type 0) or as a microsubroutine (type 1). In Figure 4.1.1 program type 1 occurs 20 times in the program and each occurrence is executed an average of 5 times for a total average execution of 100 times. It has type 0 and will be implemented with in-line sections of microcode.

Under each program type heading is a section labeled implementation, time, storage. These sections contain the information pertinent to each implementation method. The time is the average execution time (microcycles) of each implementation method. The storage is the number of control words required by each implementation

method. The strings of 1's and 0's to the right of the time and storage values specify the hardware options required by each implementation method. In Figure 4.1.1 the third implementation method of program type 2 has a time of 7 microcycles, requires 10 words of control storage and uses hardware options 1, 2, 4, and 7.

The general optimization problem is to select one implementation method for each program type so that certain cost performance criteria are satisfied. The number of program types specified for PTL is approximately 120. As was pointed out earlier, the number of possible combinations is very, very large. An exhaustive search of the total implementation space is simply out of the question.

The cost function is non-linear. The cost of a hardware option may be dependent upon the choice of other hardware options. In addition, if two or more implementations for different program types use the same hardware options, the cost of the hardware options appear only once. Thus, the choice of an implementation method will induce only a control storage cost if the hardware options required by the current implementation method were required for implementation of other program types chosen earlier.

The general problem has been outlined above. The method used to efficiently select the optimum hardware configuration will be presented in the next section.

| | HARDWARE OPTIONS | | COST | MICRO-BITS | WORD FORM |
|---|---|---|---|---|---|
| 1 | Test for Zero | $\overline{5}$ | 300 | 1 | S1 |
| | Test for Zero | 5 | 380 | 2 | S1 |
| 2 | Count by One | | 400 | 1 | S2 |
| 3 | Fast Local Storage | | 450 | 0 | S3 |
| 4 | Double Byte Read/Write | | 500 | 2 | S1, S3 |
| 5 | Full Word Alu | | 1000 | 0 | S2 |
| 6 | Floating Add/Subtract | | 2000 | 3 | S1, S2 |
| 7 | Floating Multiply | | 4000 | 1 | S3 |

Word form w = 3

SUBROUTINE REQUIREMENTS

Words to Branch 2 Words to Return 1

7 6 5 4 3 2 1

Prog. Type - 1 Freq. Occ. - 20
Freq. Exec. - 5 Type - 0

| Implementation | Time | Storage | |
|---|---|---|---|
| 1 | 3.5 | 3 | 1 0 0 0 1 0 0 |
| 2 | 4.0 | 4 | 0 1 0 0 0 0 0 |
| 3 | 5.0 | 5 | 0 0 0 0 0 0 0 |

Prog. Type - 2 Freq. Occ. - 10
Freq. Exec. - 6 Type - 0

| Implementation | Time | Storage | |
|---|---|---|---|
| 1 | 5.0 | 15 | 1 1 1 0 1 0 1 |
| 2 | 6.5 | 10 | 1 1 0 1 1 1 0 |
| 3 | 7.0 | 10 | 1 0 0 1 0 1 1 |
| 4 | 8.0 | 9 | 0 0 0 0 0 0 0 |

Prog. Type - 3 Freq. Occ. - 30
Freq. Exec. - 3 Type - 1

| Implementation | Time | Storage | |
|---|---|---|---|
| 1 | 20.0 | 40 | 1 1 0 0 0 0 0 |
| 2 | 25.0 | 35 | 1 0 0 0 1 0 0 |
| 3 | 30.0 | 30 | 0 0 0 0 0 0 0 |

Figure 4.1.1 Input Data For Optimization Program

## 4.2 General Approach

The general approach is to solve a different problem. The implementation state space is very large because there are a large number of program types and each program type may have several implementation methods. If the number of hardware options is twenty or less, the number of combinations of hardware options (the hardware option state space) is $10^6$ which is at least a factor of $10^{10}$ smaller than the implementation state space. Twenty hardware options for central processing units is a conservative estimate. It would be very nice to search the hardware option state space instead of the program type implementation state space.

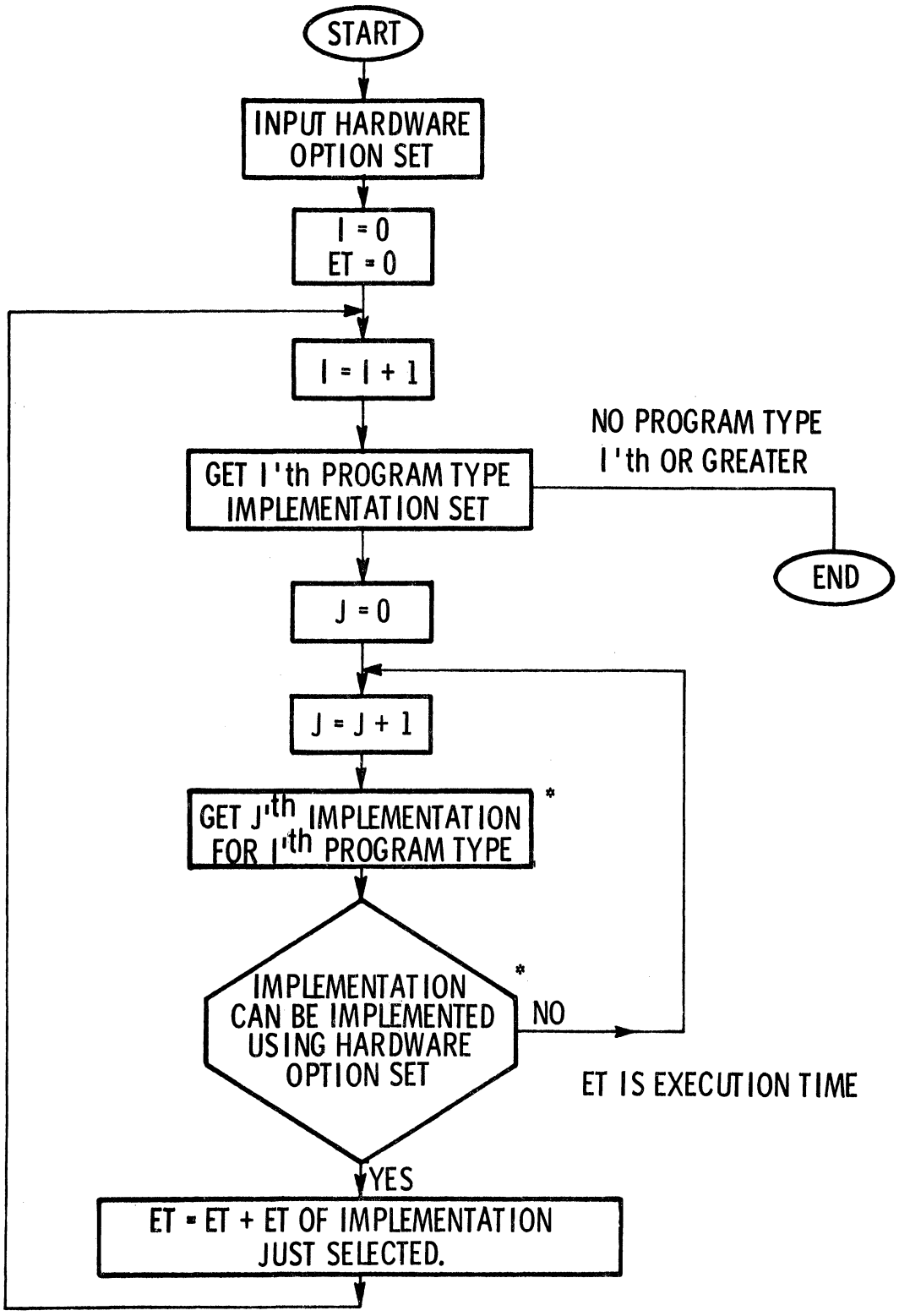
In the rest of the discussion the term implementation point will be used to describe a point in the implementation state space and hardware point will be used to describe a point in the hardware option state space. An implementation point consists of one implementation method for each of the program types. An implementation point has a single-cost which is the cost of the hardware options, the hardware base, and the control storage required by the implementation methods. The above statement assumes the existence of frequency of execution values and frequency of occurrence values for each program type.

A hardware point consists of one of the possible combinations of hardware options. Each hardware point has a specific cost which is

the cost of hardware options specified by the hardware point. Each hardware point may have many implementation points associated with itself.

It is possible to partition the implementation state space such that each partition contains implementation points which require the hardware options associated with a particular hardware option state space point. The cost of each implementation point in a partition is the cost of the hardware options associated with the partition plus the cost of control storage for the particular implementation point. Thus, it is possible to compute bounds on large numbers of implementation points (those of a partition) by using the cost of a hardware point (associated with the partition) and a bound on storage cost. It is also possible to bound the performance of each hardware point. The best performance which can be achieved under a particular hardware point can be easily calculated using the fastest implementation method of Figure 4.2.1. The method is simply one of picking the implementation with the smallest execution time for each program type which may be implemented using the hardware options of that hardware point.

Up to now it has been pointed out that the hardware option state space points partition the implementation state space points, and that a bound on the cost and performance of each hardware point may be calculated. The approach which will be taken in the optimization is to try to eliminate one point in the hardware option state space and thus many




*IMPLEMENTATIONS STORED IN ORDER, MINIMUM EXECUTION TIME FIRST, ONE IMPLEMENTATION FOR EACH PROGRAM TYPE REQUIRES NO HARDWARE OPTIONS.

Fastest Implementation Method
Figure 4.2.1

points in the implementation state space. To be effective the elimination of a point in the hardware state space should occur without an examination of the implementation state space points associated with the hardware point.

The optimization procedure will be presented in the next section. Before presenting the final optimization procedure, an exhaustive search technique will be explored This is done to give insight into the final optimization method, since that method is a modification of the two exhaustive search procedures which will be presented below Throughout the discussion it should be remembered that performance (Equation 3.2.12) is the reciprocal of the average execution time (Equation 3.2.11).

The methods to be explained below are exhaustive searches over a state space reduced from a size of $10^{15}$ or greater to a size of $10^{5}$ or less. The first procedure assumes that control storage is free. Thus, the only costs are hardware costs. This procedure is shown in Figure 4.2.2. The second procedure considers control storage costs and is shown in Figure 4.2.3.

The procedure of Figure 4.2.2 assumes that the only cost is hardware costs (control memory is free). Then for any fixed combinations of hardware options the cost is fixed. Since the hardware cost is fixed, the selection of each program type implementation is dependent only upon the execution time of the implementation and its requirements

for hardware options. For a fixed set of hardware options, the implementation method for each program type should be the fastest that can be implemented under the given hardware options. Therefore, the fastest implementation methods for a fixed cost (the cost of the hardware options) have to be chosen.

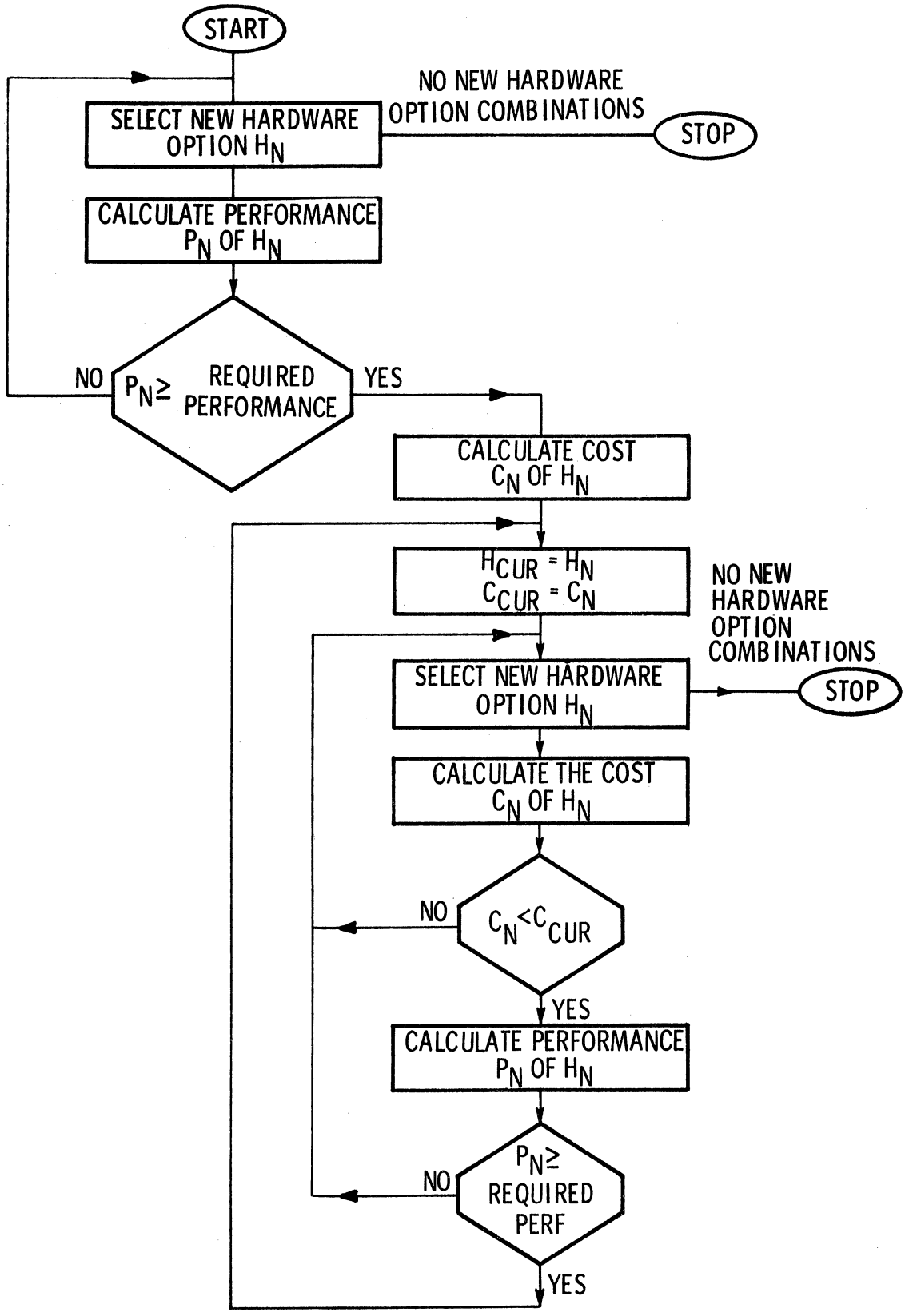
The search method of Figure 4.2.2 examines every possible set of hardware options. It examines each hardware point and calculates the execution time using the method (fastest implementation) outlined above until a hardware point is found which satisfies the performance requirement. Once a hardware point which satisfies the performance requirement is found, that hardware point becomes the current hardware point. The current hardware point is that hardware point examined up to now which has the lowest cost of any hardware point meeting the performance requirement. After finding the first hardware point, the search is continued, but each new hardware point has its cost calculated before its performance. If the cost is greater than or equal to the current hardware point, the present hardware point is discarded and a new one is selected. If the cost is less than the cost of the current hardware point, performance of the new hardware point is computed. If the performance is not acceptable the hardware point is discarded and a new set is selected. If the performance is acceptable the new hardware point becomes the current hardware point and a new point to test is selected. This process continues until all the

hardware points have been tested.

The cost is tested before the performance after finding an initial solution because the hardware option cost can be calculated much more rapidly than the performance. Since many hardware options can be eliminated by cost, it seems reasonable to test the cost before the performance.

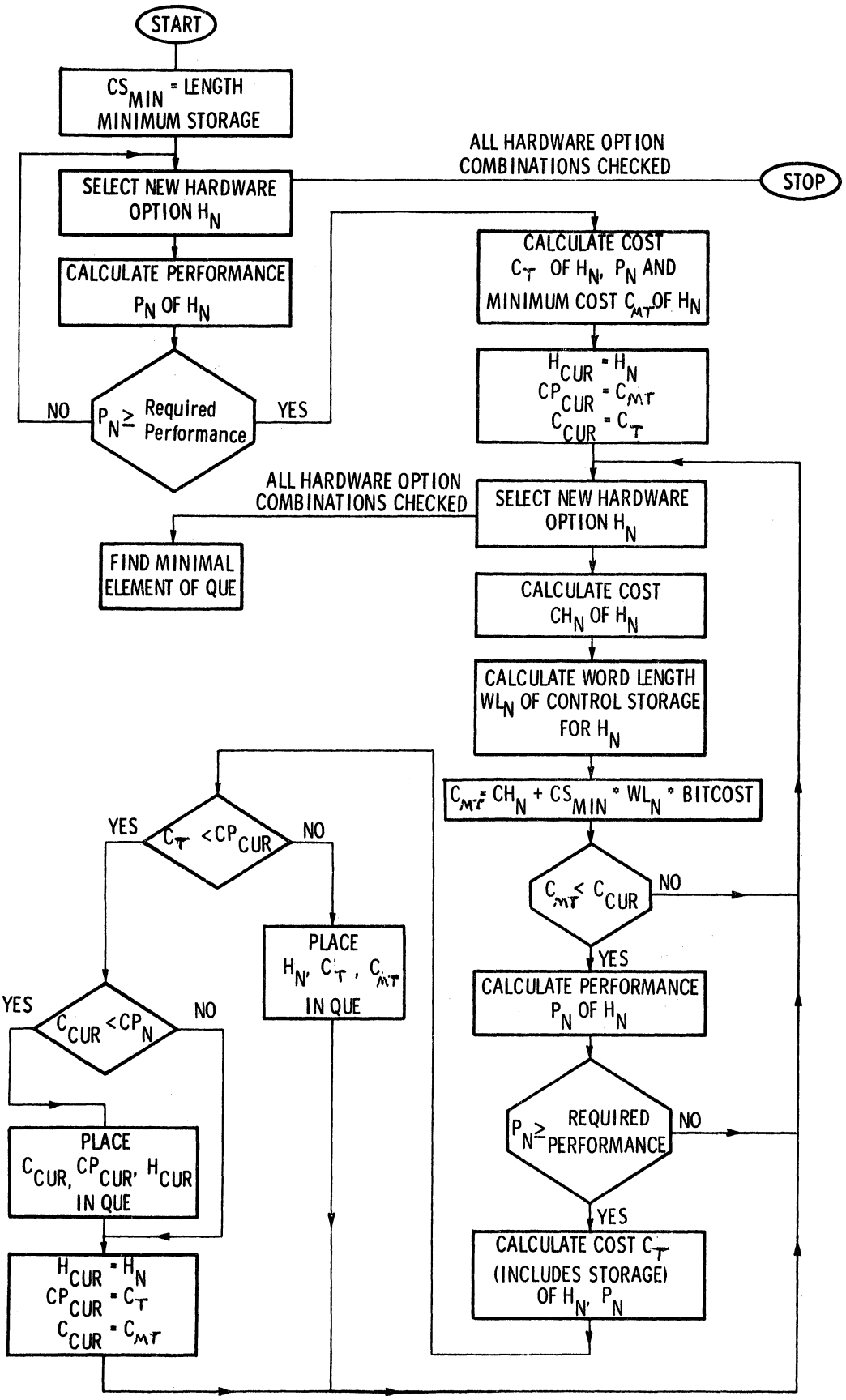
The procedure of Figure 4.2.2 assumes that control memory is free or that only one size of control memory is available. The problem becomes slightly more complicated if the cost of control memory is considered. The goal is still to eliminate hardware points and thus many implementation points without examining every member (set of implementations) of each partition. What is done in the method flowcharted in Figure 4.2.3 is to bound the cost of the implementation points in a partition and try to eliminate hardware points on the basis of the bounded costs. The performance test does not have to be modified from Figure 4.2.2 since the fastest implementation method still computes the minimum execution time which may be achieved under a particular hardware point. Thus, if the execution time is not satisfactory no other set of implementations which are feasible under the set of hardware options of the hardware point will be satisfactory.

The procedure of Figure 4.2.3 is similar to Figure 4.2.2. Every possible hardware option set is examined. Points from the hardware option space are selected and the fastest implementation is




Exhaustive Search Technique Control Memory Free
Figure 4.2.2

computed until a set is found which satisfies the performance requirement. The cost of that implementation set (hardware point) is computed. The search is then continued but the cost of the new point is first calculated and tested against the cost of the current hardware option set. The cost of the new point is the cost of the hardware plus the cost of storage. At this point, the optimal implementation point in the partition of implementation points defined by the hardware option point is not known. Therefore, the exact cost of control storage associated with the hardware option is not known. What is done is to use the lower bound on the number of words of control storage. If the cost of the new hardware point plus the lower bound on storage cost exceeds the cost of the current hardware option set the hardware option set is discarded. If the lower bound on cost ($C_{MT}$ of Figure 4.2.3) is below the current cost, the fastest performance is calculated. If the performance does not satisfy the performance requirement (reciprocal of execution time), the hardware point is discarded. If the performance requirements are satisfied, the actual cost ($C_T$ of Figure 4.3.3) of the fastest implementation is calculated. If the cost is less than the cost of the current hardware point, the new hardware point replaces the current hardware point. However, the previous current hardware point may or may not be discarded. The previous current hardware point is discarded only if its lower bound in cost is greater than the cost of the new current hardware point. If this is not the case the previous




Exhaustive Search Technique
Figure 4.2.3

current hardware point is placed in a storage queue (called QUE). When a new current hardware point is found, each member of QUE is tested to see if it can be eliminated. A member of QUE is eliminated if its lower bound cost is greater than the cost (of the fastest implementation) of the current hardware point. This procedure is continued until all points in the hardware option space have been examined. Assume that QUE never overflows. After examining all points in the hardware option state space there may be several hardware points in QUE. Each hardware point in QUE has a $C_{MT}$ cost which is below the $C_T$ cost of any element in QUE and a $P_N$ (fastest implementation) which is acceptable. The hardware points in QUE define a set of implementation points and some technique must be developed to select the optimal implementation point. That function is represented by the box FIND MINIMAL ELEMENT OF QUE of Figure 4.2.3. The flowchart for the procedure is given in Figure 4.2.4.

The selection of an optimal implementation point (Figure 4.2.4) from the elements of QUE requires two procedures. One procedure is the slope selection algorithm which selects that implementation point from an implementation point partition defined by a hardware point which achieves a required performance and has minimum cost. The second procedure eliminates the hardware points in QUE using the slope selection algorithm and conditional tests until only the optimal solution is left. The internal workings of the slope selection procedure

will be explained in the next section.

The combined procedure (Figure 4.2.4) works as follows. Using the slope selection algorithm, the cost ($C_{TB}$) of the optimal implementation set ($I_{TB}$) of the first hardware point ($H_{TB}$) in QUE is calculated. Remove the first hardware point in QUE. If QUE is empty the optimal solution has been found. If not, remove any element in QUE which has a $C_{MT}$ cost greater than $C_B$. If QUE is empty the optimal solution has been found. If not, compute the cost ($C_{TB}'$) of $I_B'$ of $H_B'$ (the first hardware point in QUE). The optimal implementation (either $I_{TB}$ or $I_{TB}'$) is selected. This procedure is then repeated until QUE is empty (using a combination of the $C_{MT}$ test and $C_{TB}$ test). The remaining $I_{TB}$ is the optimal solution.

The optimization procedure of Figure 4.2.3 has been presented to aid the understanding of the optimization problem. The procedure used an exhaustive search technique, and although relatively fast, its performance can be improved using a modification of the Lawler-Bell algorithm. Both the Lawler-Bell algorithm and the slope selection algorithm will be discussed in the next section.

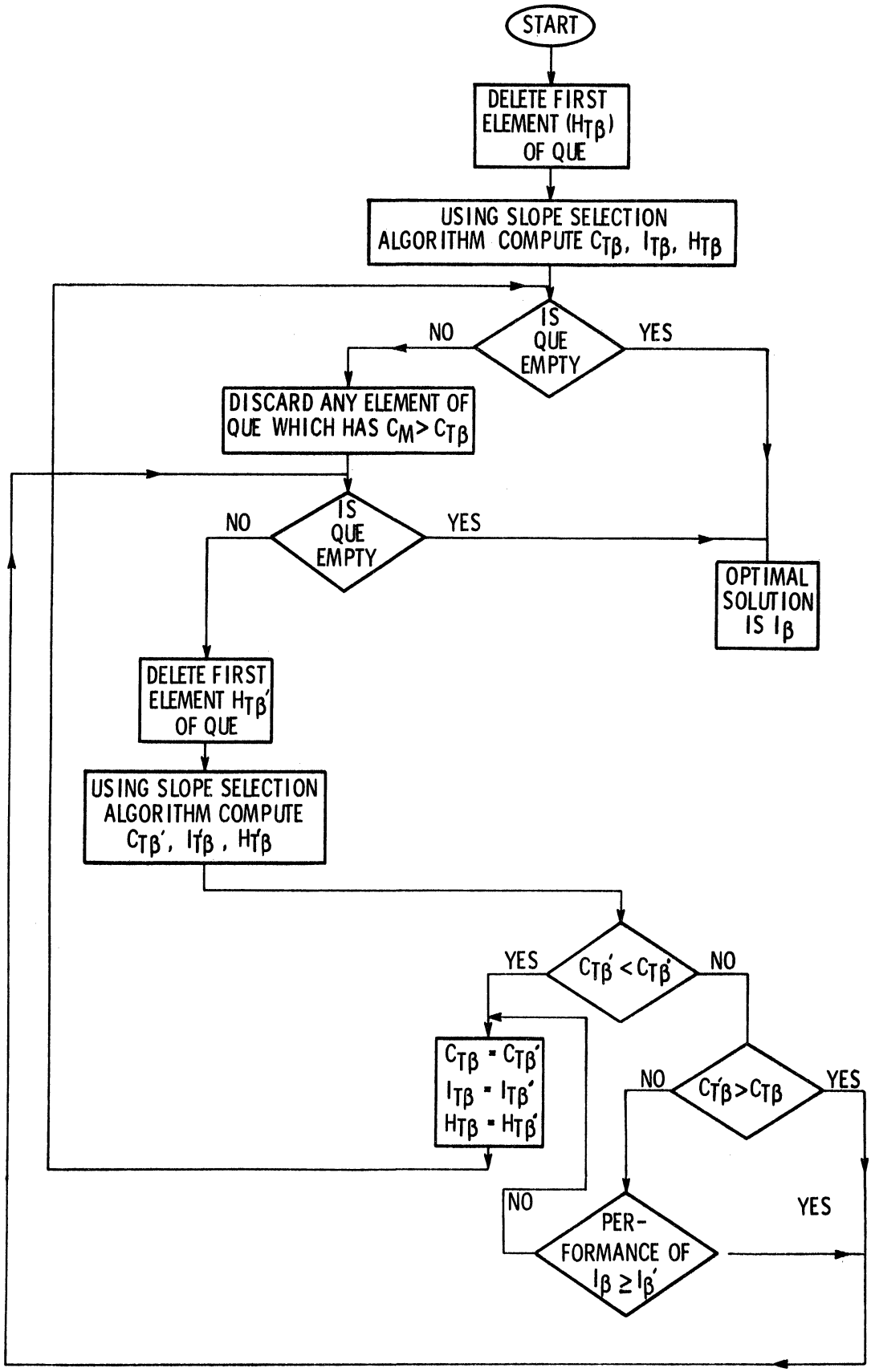

START
DELETE FIRST ELEMENT ($H_{T\beta}$) OF QUE
USING SLOPE SELECTION ALGORITHM COMPUTE $C_{T\beta}$, $I_{T\beta}$, $H_{T\beta}$
IS QUE EMPTY
NO
YES
DISCARD ANY ELEMENT OF QUE WHICH HAS $C_M > C_{T\beta}$
IS QUE EMPTY
NO
YES
OPTIMAL SOLUTION IS $I_\beta$
DELETE FIRST ELEMENT $H_{T\beta'}$ OF QUE
USING SLOPE SELECTION ALGORITHM COMPUTE $C_{T\beta'}$, $I_{T\beta}$, $H_{T\beta}$
YES
$C_{T\beta'} < C_{T\beta'}$
NO
$C_{T\beta} = C_{T\beta'}$
$I_{T\beta} = I_{T\beta'}$
$H_{T\beta} = H_{T\beta'}$
NO
$C_{T\beta} > C_{T\beta}$
YES
NO
PER-FORMANCE OF $I_\beta \geq I_{\beta'}$
YES
FUND MINIMAL ELEMENT OF QUE

Figure 4.2.4

## 4.3 Optimization Procedure

The optimization techniques (Figures 4.2.1, 4.2.2, and 4.2.3) presented in the last section converted a computationally infeasible problem ($10^{15}$ points) to one which can be handled ($10^{6}$ points) by present day computing systems. The speed of the optimization procedure of Figure 4.2.3, which is the general optimization procedure, can be further improved by applying a modification of the Lawler-Bell [L4] technique for selecting the elements of QUE.

The optimization procedure flow chart is shown in Figure 4.3.1. Before presenting proof of the validity of the technique, the procedure will be described.

The technique is one of partial enumeration. The hardware option state space is assigned a numerical ordering using Equation 4.3.1. The space is searched from no hardware options ($0 \cdots 0$), to all hardware options ($111 \cdots 1$). The numerical ordering does not, in general, correspond to an ordering by cost. If such were the case, once an acceptable performance were found, the search should be stopped since all remaining hardware option points would have a higher cost. Although the numerical ordering does not correspond to an ordering by cost, it does happen that many sub-groups of the numerical ordering have a useful cost ordering. In particular, it turns out that for each hardware point $x$ it is possible to compute the next hardware point $x'$

which could be lower in cost than x. Thus, if x has an unacceptably high cost it is possible to skip all points between x and x'. In addition, if x has an acceptable cost, it is possible to compute the best possible performance of the points from x to but not including x'. If the performance is not acceptable the points between x and x' are skipped. Thus, the major difference between the flow chart of Figure 4.2.3 and 4.3.1 is the use of partial enumeration instead of searching the entire state space. The flow chart of Figure 4.3.1 shows only the hardware point selection. The selection of the optimum implementation point for a hardware point is discussed later.

In the optimization procedure of Figure 4.3.1 the first objective is to find a feasible solution. The no hardware option case is tested first. If the no hardware option performance is not acceptable, then the all hardware option case is taken. The all hardware case is assumed to have acceptable performance since no other set of hardware could produce a performance which is better. Thus, we assume that the required performance $P_r$ is achievable (i.e., this is checked at input time). Then the hardware option state space is searched from the point $(0 \cdots 01)$ with a numerical order of 1 to point $(11 \cdots 11)$ with a numerical order of $2^{|H|} - 1$.

The main loop starts at point A of Figure 4.3.1. The value SMIN (the minimum number of control storage words required by any implementation point) is used to calculate $C_{MT}$ for the current hardware

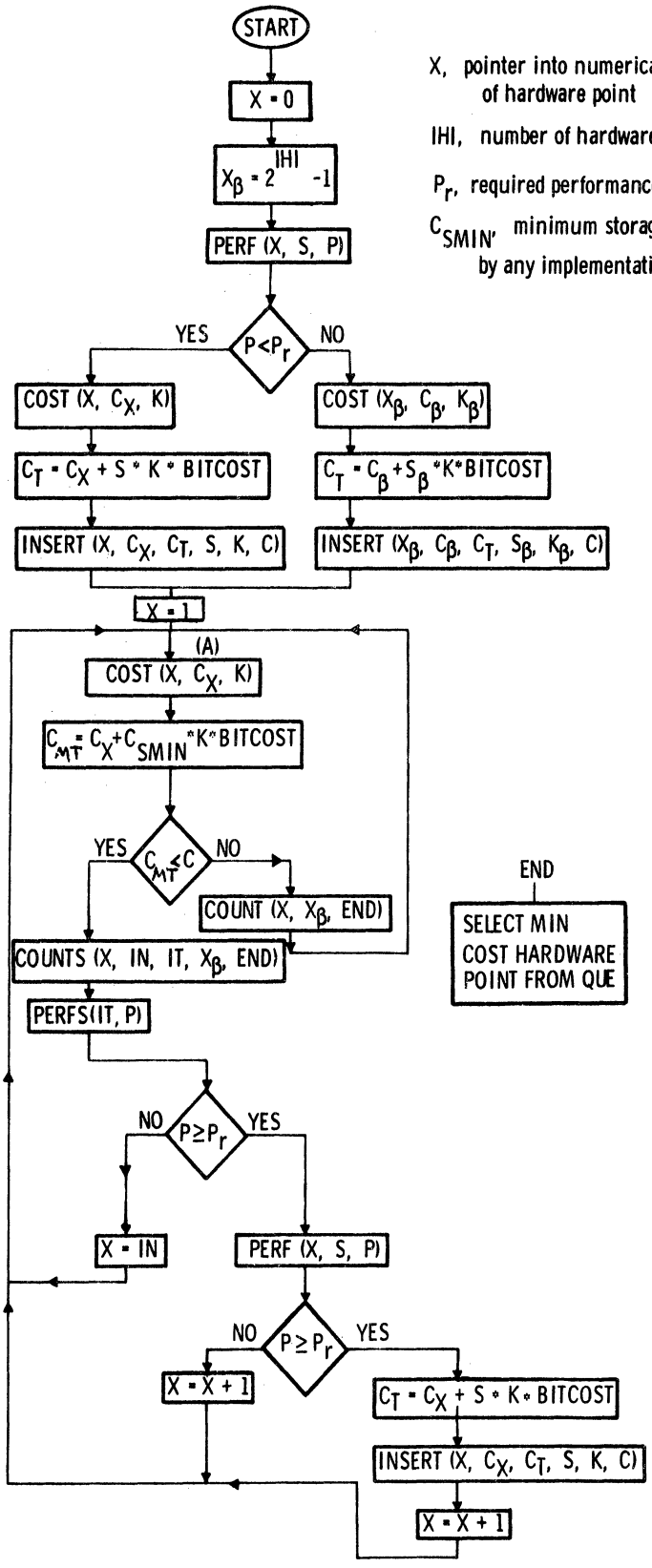
point. $C_{MT}$ is the minimum possible cost (cost of the hardware options plus the cost of SMIN words of control storage) for any implementation point defined by the current hardware option point. Although SMIN is fixed the width of the control storage words is a function of the hardware options specified by a hardware point. The value of $C_T$ for a hardware point is the cost of the hardware options specified by the hardware point plus the cost of the control storage required by the fastest implementation point defined by that hardware option point. The value of $C_{MT}$ is compared to the value of C. The value for C is the minimum $C_T$ value of any hardware point examined so far which had an acceptable performance. If $C_{MT}$ is greater than or equal to C, a jump in the numerical order to the next point lower in cost is taken. The next point is computed by COUNT(X, $X_n$, END). The routine accepts the value (numerical ordering) of the current point X and then calculates a new value for X which is the next point in the numerical ordering that could have a cost less than the cost of X. The number of points which are skipped can be considerable. If $X_\beta$ is reached, COUNT returns to END.

If the $C_{MT}$ cost of the current hardware point is less than C the routine COUNTS is called. This routine calculates IN which is the next hardware option (in numerical order) that could have a cost less than $C_{MT}$ (i.e., the same point which would have been calculated by COUNT), and the hardware option point (IT) which contains the hardware

options used by X, and by all the hardware points between X and IN in numerical ordering. Then the optimal performance (P) of IT is calculated. If P is not acceptable the current point is replaced by IN, skipping any points between X and IN. If P is acceptable then the optimal performance P of X is calculated and compared against the required performance. If the optimal performance of X is not acceptable, the next hardware point after X (X+1) is chosen and a branch to the start of the loop (A) is taken. If P of X is acceptable, $C_T$ of X is calculated and the values of X, $C_T$, S (the storage required by the fastest implementation point of X), and K (the micro-bit required by X) are placed in the queue (QUE) of possible optimums. The value of C is set equal to $C_T$ if $C_T$ is less than the previous value of C. Then the next hardware point after X (X+1) is chosen and a branch to the start of the loop (A) is taken.

The procedure of Figure 4.3.1 uses a modification of the Lawler-Bell algorithm. The modifications of the algorithms are extensive since two of the three selection rules must be completely changed. Therefore, the proof of the modified algorithm will be given from the start even though part of the discussion parallels that of the Lawler-Bell paper [L4]

Before proceeding some essential preliminaries must be discussed. The value of n(x) given by Equation 4.3.1 introduces a numerical ordering on the hardware option state space. It is also




Optimization Procedure
Figure 4.3.1

PERF (X, S, P) S, P computed

P best performance for X

S storage required by X to achieve P

PERFS (IT, P) P computed

P best performance for IT

COUNT ($X, X_{\beta}$, END)

Algorithm 4.3.2 is used to compute new value of X from current X.

Return to END occurs if $X > X_{\beta}$

COUNTS (X, IN, IT, $I_{\beta}$, END)

Algorithm 4.3.2 is used to compute value of IN from X, IT = IN - 1

Return to END occurs if $IN > X_{\beta}$

COST ($X, C_X, K$) calculate $C_X, K$

$C_X$ is the hardware option cost of X

K is the number of microbits for hardware option point X

INSERT ($X, C_X, C_T, S, K, C$)

The current hardware point X, and its hardware options cost $C_X$, total cost $C_T$, storage S, and microbits K, are stored in the QUE of possible optimums. The value returned (C), is the best cost of any item in QUE.

Figure 4.3.1 (Continued)

possible to consider a vector partial ordering between elements of the hardware option state space. Each element $x$ of the hardware option state space is represented as a binary vector. Two vectors, $x$ and $y$ have a vector partial ordering if the relationship of Equation 4.3.2 is satisfied. For example $x \leq y$ if $x = (100)$ and $y = (101)$ but if $w = (100)$ and $z = (001)$ there is no vector partial ordering between $w$ and $z$.

4.3.1 Let $x = (x_N, x_{N-1}, \ldots, x_1)$

$$n(x) = 2^{N-1} x_N + 2^{N-2} x_{N-1} + \ldots + 2^1 \cdot x_2 + 2^0 \cdot x_1$$

4.3.2 $\quad x \leq y \quad$ if $\quad x_i \leq y_i \quad$ for $\quad i = 1, 2, 3, \ldots, N$

A vector partial ordering implies a numerical ordering (Equation 4.3.3), but not vice versa (Equation 4.3.4).

4.3.3 $\quad x \leq y \implies n(x) \leq n(y)$

4.3.4 $\quad n(x) \leq n(y) \not\Rightarrow x \leq y$

However, there is a very interesting relation between numerical ordering and vector partial ordering. Assume that the binary $n$ vectors between $x_0 = (000 \cdots 00)$ and $x_{2^N-1} = (111 \cdots 11)$ are ordered by the numerical ordering of Equation 4.3.1. It is very simple to compute the next binary vector $x'$ (in numerical order) after any vector $x$

which is not in a partial ordering relation to x. Thus x' is the first vector following x which has the property that $x \not\leq x'$. The algorithm for computing x' from x is given below (Figure 4.3.2). The algorithm is the one used by COUNTS (Figure 4.3.1) where x is X, and x' is IN.

1. subtract 1 from x
2. logically or x with x-1 to obtain x'-1
3. add 1 to obtain x'

Algorithm for Calculating x' from x
Figure 4.3.2

There are three procedures in the optimization which must be justified. First, if the minimum cost of hardware option point x is greater than the current cost, the hardware option points between x and x' (x' is computed from x using algorithm of Figure 4.3.2) are skipped. Second, if the optimum performance achieved by the union of hardware options of x, and the hardware options of the hardware points between x and x' do not satisfy the performance requirements, the hardware option points between x and x' are skipped. The third procedure is used to select the minimum possible cost for a particular hardware point (i.e., minimize storage costs).

The first two will be proven now, and are related to the Lawler-Bell algorithm. The third is a new optimization method and will be

explained separately.

The proof of the first procedure requires the definition of the minimum cost of a hardware point $C_m(x)$. The value $C_m(x)$ is that represented as $C_{MT}$ in the flow chart of Figure 4.3.1. The minimum cost $C_m(x)$ is essentially a lower bound on the cost of the hardware options specified by $x$, plus the base cost, plus the minimum control storage cost. The minimum words of control storage is constant for all hardware points and is the sum of the minimum length storage for each set of program type implementations. The width of each control word varies from hardware option point to hardware option point. The width is the sum of the base microbits and the number of microbits required to control each hardware option of the hardware option state space point.

Definition 4.3.1

Minimum Hardware Point Cost; $C_m(x)$: The minimum cost of hardware point is the cost of the base hardware and hardware options specified by hardware option point $x$, plus the cost of the minimum possible length of control storage.

Lemma 4.3.1 If $x$ and $y$ are two hardware option points and there is a partial ordering of $x$ and $y$ such that

$$x \leq y, \text{ then}$$

$$C_m(x) \leq C_m(y)$$

Proof: Observe that if $H(x)$ is the hardware option cost of $x$, and if $x \leq y$, then $H(x) \leq H(y)$.

Observe next that if $CS(x)$ is the minimum storage cost of $x$, and if $x \leq y$, then $CS(x) \leq CS(y)$.

Hence

$$C_m(x) = H(x) + CS(x) + c_b \leq H(y) + CS(y) \cdot c_b = C_m(y)$$

where $c_b$ is the base hardware cost.

Definition 4.3.2

Hardware Option Span; $h(x, x')$: The hardware option span is the representation of hardware options specified by $n(x)$, $n(x+1)$, $n(x+2)$, ..., up to but not including $n(x')$.

Definition 4.3.3

Best Performance; $P(x)$: The Best Performance $P(x)$ gives the value of the optimum performance which may be achieved using the hardware option set represented by $x$.

Lemma 4.3.2: Let $n(x') > n(x)$ and given a desired performance $P'$; if $P(h(x, x')) < P'$ then for

each $x''$ such that $n(x) \leq n(x'') < n(x')$

$$P(x'') < P'$$

Proof: The hardware option set represented by $x''$ is a subset of the hardware option set specified by $h(x, x')$. But, the addition of a hardware option can never reduce performance.

$\therefore \quad P(x'') < P'$

Definition 4.3.4

Next Point; NP(x): The next point function NP(x) takes the representation x of a hardware option set and calculates the representation x' of another hardware option set such that

1) $n(x) < n(x')$
2) $x \not\leq x'$
3) no $x''$ exist such that
   a) $n(x) < n(x'') < n(x')$ and
   b) $x \not\leq x''$

The next point function is calculated by the algorithm specified in Figure 4.3.2. The next point function is used to skip sets of hardware option (those between x and x') if certain conditions are satisfied. The conditions are given below and they are the rules used in the optimization procedure which is presented in Figure 4.3.1.

The value $P_r$ used in Rule 4.3.2 and Theorem 4.3.2 is the required performance. The value $C_m$ used in Rule 4.3.1 and Theorem 4.3.1 is the minimum cost hardware point of those hardware points examined so far that had acceptable performance. The value $C$ is the cost of the hardware options plus the cost of control storage for the fastest implementation point specified by the minimum cost hardware point.

Rule 4.3.1 If $C_m(x) > C$ skip to hardware option

$$x' = NP(x)$$

Rule 4.3.2 If $P(h(x, x')) < P_r$ skip to hardware option

$$x' = NP(x)$$

Theorem 4.3.1 The use of Rule 4.3.1 will eliminate only nonoptimal hardware option points.

Proof: by Definition 4.3.4 any hardware option $x''$ between $n(x)$ and $n(x')$ satisfies the partial ordering relation $x \leq x''$

$\therefore$ by Lemma 4.3.1

$$C_m(x'') \geq C_m(x)$$

$\therefore$ $C_m(x'') > c$

$\therefore$ therefore there is no hardware point between $n(x)$ and $n(x')$ which would cost less than current lowest cost $c$.

Theorem 4.3.2 The use of Rule 4.3.2 will eliminate only non-optimal hardware option points.

Proof by Lemma 4.3.2 for each x'' such that

$$n(x) \leq n(x'') \leq n(x')$$

$$P(x'') < P_r$$

$\therefore$ any x'' would have a performance which is less than the minimum acceptable performance.

Both Rule 1 and Rule 2 skip sets of hardware option points without examining any of the members of the set. Rule 1 uses a cost criteria and Rule 2 uses a performance criteria. With Rule 1, the set of hardware points between n(x) and n(x') is skipped if the minimum cost ($C_m(x)$) of x is greater than the best cost (C) hardware point the optimization has currently found. The jump from x to x' is taken since all hardware points between n(x) and n(x') have a minimum cost greater than $C_m(x)$, if x' is calculated using the algorithm of Figure 4.3.2. The algorithm of Figure 4.3.2 identifies in an efficient manner sections of the hardware option point space which are ordered by cost. With Rule 2, the set of hardware points between n(x) and n(x') is skipped if the best performance of any of the hardware option points specified by n(x), n(x+1), ..., n(x'-1) is not an acceptable performance. The test used by Rule 2 calculates a bound on the performance of the

hardware points between n(x) and n(x'). If the bounded performance is not acceptable, none of the hardware points between n(x) and n(x') will have an acceptable performance.

The second optimization technique is used to obtain the minimal cost set of implementations for a hardware point which is known to have a feasible (in terms of performance) set of implementation methods. The problem can be stated as follows. Given a hardware point and a set of the fastest implementations that are feasible (achieve the desired performance goals) it may be possible to reduce the cost of the solution (reduce storage) and still satisfy feasibility.

Figure 4.3.3 is an example of such a problem. For a fixed hardware point (01011) the minimum execution time implementations are checked. The resulting cost is the cost of the hardware plus the cost of the control storage. The hardware cost is fixed since the hardware options are fixed, but the cost of control storage could be reduced. The storage required for a program type implementation can be reduced if there are implementations which can be implemented using the current set of hardware options but require less storage than the checked implementation. Such implementations are marked with an x in Figure 4.3.3. Thus, storage can be saved but at the cost of performance since implementations which require additional microcycles are being utilized.

Since control storage is available in unit sizes, the reduction

| Program | Type Implementation | Time | Storage | Hardware Options $H_1$ | $H_2$ | $H_3$ | $H_4$ | $H_5$ | |
|---|---|---|---|---|---|---|---|---|---|
| $P_1$ | 1 | 2 | 3 | 0 | 1 | 0 | 1 | 1 | ✓ |
| | 2 | 3 | 6 | 1 | 0 | 1 | 1 | 0 | |
| | 3 | 3.5 | 5 | 1 | 1 | 0 | 0 | 0 | |
| | 4 | 4 | 7 | 0 | 0 | 0 | 0 | 0 | |
| $P_2$ | 1 | 2 | 7 | 0 | 1 | 0 | 1 | 1 | ✓ |
| | 2 | 2.5 | 4 | 0 | 1 | 0 | 1 | 0 | x |
| | 3 | 4 | 8 | 0 | 0 | 0 | 0 | 0 | |
| $P_3$ | 1 | 3 | 7 | 1 | 0 | 1 | 0 | 0 | |
| | 2 | 3.5 | 8 | 1 | 0 | 0 | 1 | 0 | |
| | 3 | 4 | 9 | 0 | 1 | 0 | 0 | 1 | ✓ |
| | 4 | 5 | 6 | 0 | 0 | 0 | 0 | 0 | x |
| $P_4$ | 1 | 2 | 7 | 0 | 1 | 0 | 0 | 1 | ✓ |
| | 2 | 3 | 5 | 0 | 1 | 0 | 0 | 0 | x |
| | 3 | 3.5 | 5 | 0 | 0 | 1 | 0 | 1 | |
| | 4 | 4 | 3 | 0 | 0 | 0 | 0 | 0 | x |

Fastest Implementation and Alternates
for Hardware Point X = 01011

Figure 4.3.3

in storage should reduce the total storage requirements in integral units. The goal will be to achieve the desired reduction in control storage with a minimum reduction in performance. It may be the case that there is no solution which saves a page of control storage and still achieves the desired performance goal.

The problem is shown graphically in Figure 4.3.4. The fastest set of implications is represented by $p_1$. The horizontal dotted line represents the maximum acceptable execution time. The vertical dotted line through $p_2$ represents the reduction in storage required to reduce the storage cost by one page unit. Each of the line segments from $p_1$ to $p_2$ represents the reduction in storage and the resulting increase in execution time if the fastest implementation for a program type is replaced by another implementation which requires less storage but also increases execution time. The $p_2$ point of Figure 4.3.4 is the cost performance point resulting from the reduction in control storage of one page.

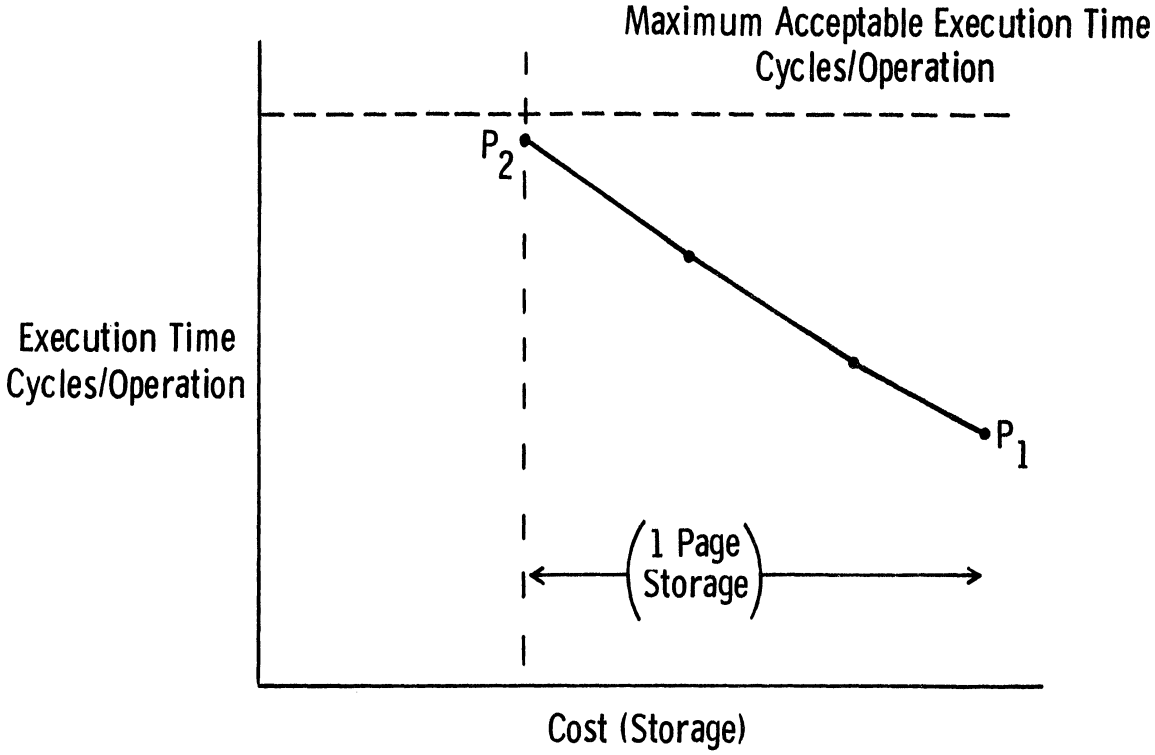
The graphical illustration of Figure 4.3.4 suggests an optimization technique. Let $\Delta t^j_{\ell k}$ represent the increase in execution time and $\Delta s^j_{\ell k}$ the reduction in control storage which results if the $\ell$th implementation method for the jth program type were replaced by the kth implementation method for the jth program type. It seems reasonable to select among alternatives the program type j with the smallest $\frac{\Delta t^j_{\ell k}}{\Delta s^j_{\ell k}}$ where $\ell$ is the fastest feasible implementation for the jth

program type and k is allowed to range over all other feasible implementations for the jth program type The $\ell$th implementation for the jth program type would be replaced by the kth implementation for the jth program type. Then the program type (call it m) which had the second smallest $\Delta t/\Delta s$ value would be chosen. This procedure would continue until either the desired savings in control storage is achieved or the performance is degraded below an acceptable level.

The method is indeed optimal if there is only one replacement candidate for each program type. If there are two or more candidates the method will not guarantee an optimal solution. The counter example is shown graphically in Figure 4.3.5. What occurs is that for program type 1, implementation 3 (1, 3 in Figure 4.3.5) has a slightly worse slope than implementation 2 (1, 2), but it saves more total storage. Thus, one additional implementation must be replaced if implementation (1, 2) is used, and the resulting average slope is worsened.

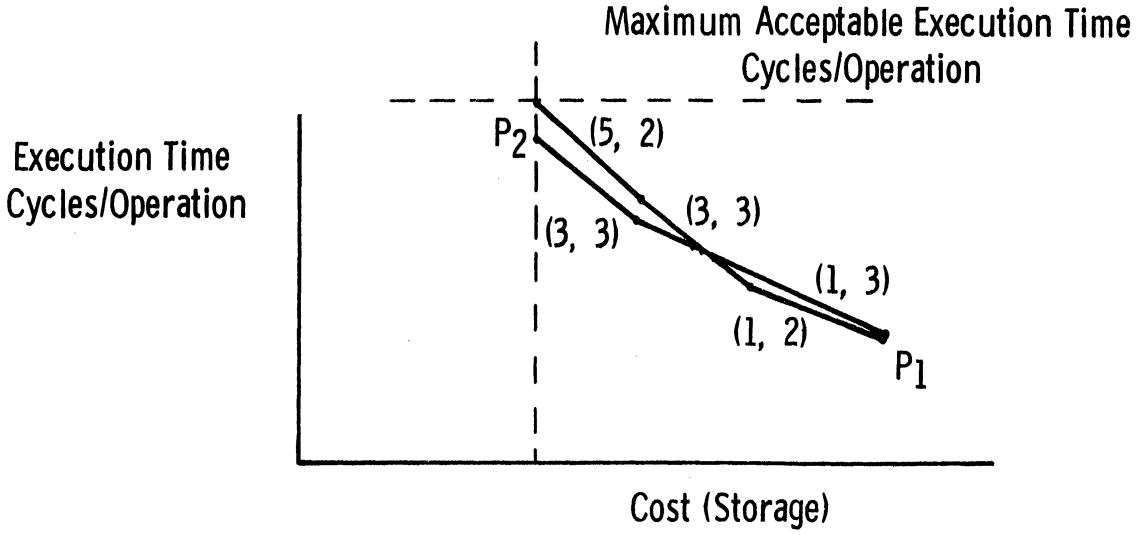
What is done is to use a slope selection procedure but to look at second third, etc., differences.

For each program type, the best first difference slope is calculated. The first difference is the smallest $\frac{\Delta t^j_{ik}}{\Delta s^j_{ik}}$ between the fastest feasible implementation (call it i) for the jth program type and the other feasible implementations. If implementation k has the best $\Delta t^j_{ik}/\Delta s^j_{ik}$, it is the implementation which would cause the least reduction in performance per unit savings in control storage for that program type.




Graphical Representation of Cost Performance Tradeoff

Figure 4.3.4




First Difference Selection Example
Figure 4.3.5

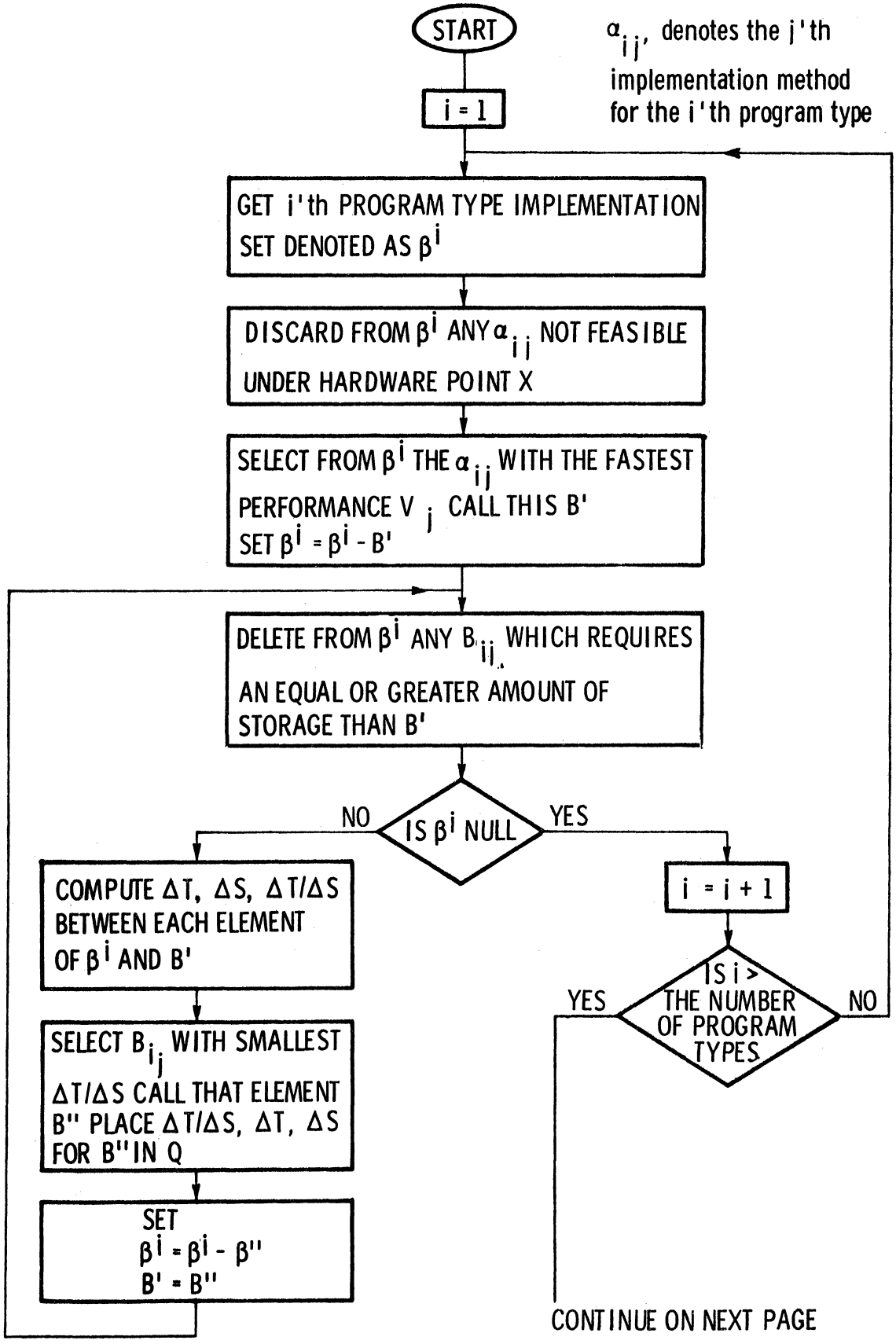
After the best first difference for a program type is found the best second difference is calculated. The procedure is the same as that for the first difference except that $\frac{\Delta t^j_{km}}{\Delta s^j_{km}}$ is calculated. Thus the fastest implementation (i) is replaced by implementation k in the $\Delta t$ and $\Delta s$ calculations. The third difference is then calculated, etc. The calculations stop when there are no additional implementations for the jth program which could cause a savings in control storage. The flow chart for the slope selection method is shown in Figure 4.3.6. The first step is to calculate the first, second, etc. differences for each program type. The first, second, etc. differences $(\Delta t^j_{ik}/\Delta s^j_{ik})$ each form a triple $(\Delta t^j_{ik}/\Delta s^j_{ik}, \Delta t^j_{ik}, \Delta s^j_{ik})$ which are placed in a queue which is denoted as Q in the flow chart of Figure 4.3.6. The values in Q are ordered with the first element in Q having the smallest $\Delta t/\Delta s$ value, the second element in Q having the second smallest $\Delta t/\Delta s$ value etc. This means that the first difference for program type j will occur before the second difference for program type j in Q. However, the second difference for program type j could have a smaller $\Delta t/\Delta s$ value than the $\Delta t/\Delta s$ value for the first difference of program type m and would occur before the first difference of program type m in Q.

The rest of the optimization is elegantly simple and is accomplished by taking the top item out of the Q and keeping a running sum on dT and dS. The variables dT and dS are the increase in

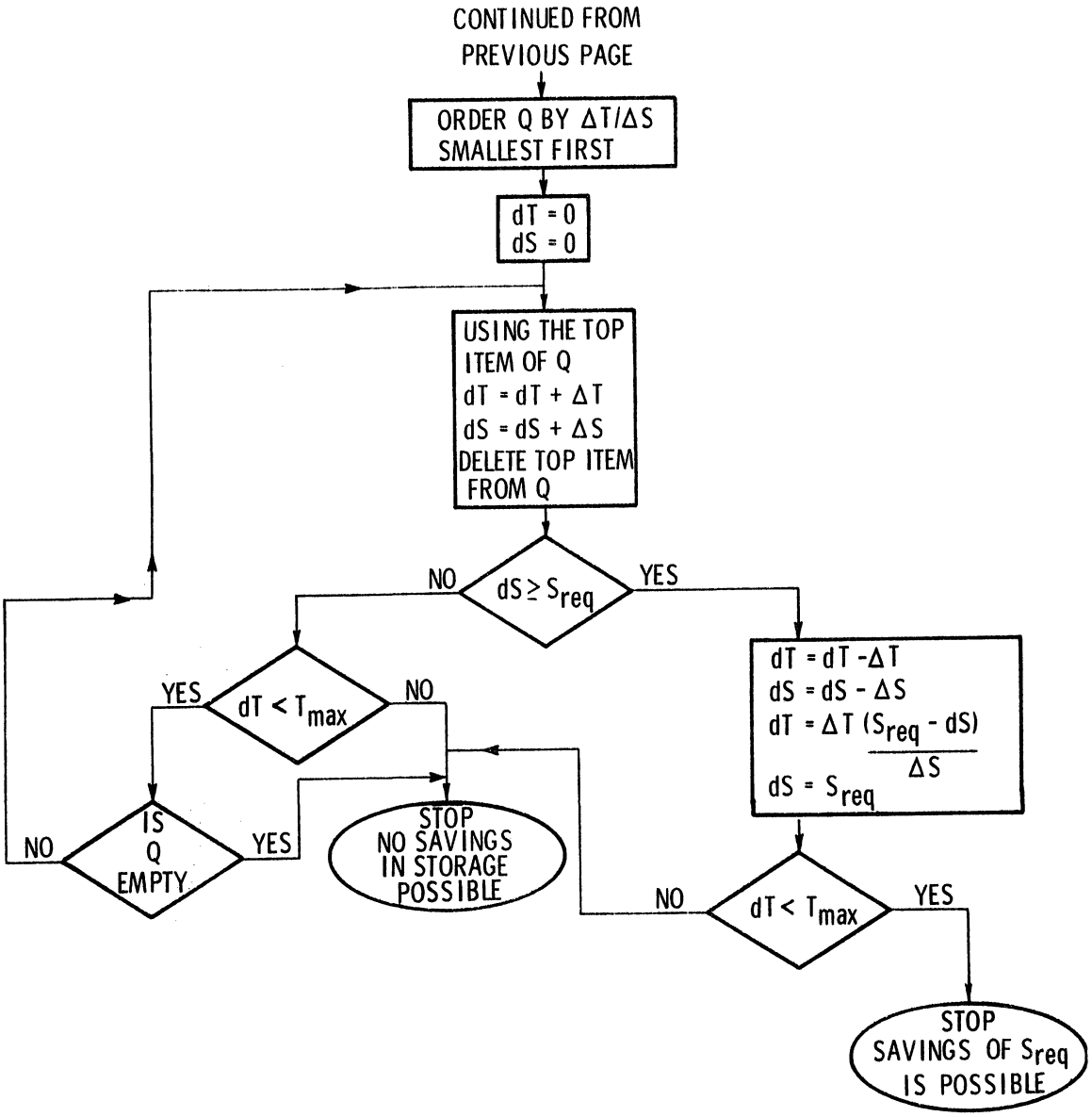
execution time and the reduction in control storage achieved after each iteration of the procedure of Figure 4.3.6. If one or more pages of storage can be saved without increasing the execution time above that required, a cost reduction has been effected. If dT degrades the execution time before dS reaches a value which saves a page of storage, no cost reduction can be achieved and the fastest set of implementations are selected for that particular hardware point.

The $\Delta S^{j}_{ik}$ values generally represent a savings of several words of control storage. Therefore, the value of dS (after several implementation replacements) would be expected to exceed slightly the value required to reduce the control storage by exactly an integral number of pages. The optimization procedure of Figure 4.3.6 avoids that problem by using a fractional part of the storage saved by the last replacement implementation such that dS does represent an integral number of control storage pages. The value of dT is also appropriately modified. Such a procedure is reasonable since a mixture of the fastest implementation and the replacement implementation could be used to very nearly achieve the desired savings in control storage.

The use of first, second, third, etc. differences provides an automatic selection process once the differences have been calculated, stored, and ordered in a queue. The first differences for a program type will always exhibit a better $\Delta t/\Delta s$ slope than the second difference,




Slope Selection Method
Figure 4.3.6




Slope Selection Method
Figure 4.3.6 (Continued)

the second difference $\Delta t/\Delta s$ will be better than the third difference, etc. Therefore, the best replacement implementation for a program type will always be chosen first. The second best replacement implementation will only be chosen if the second difference for that implementation is at the top of the queue before the selection process is complete. If the second difference is at the top of the queue, the increase in execution time per unit of control storage reduction is less than for any remaining replacement implementation for other program types. The second difference for one program type may have a better $\Delta t/\Delta s$ than the first difference for some other program type. Thus, the second best replacement implementation, not the best replacement implementation, should be and is chosen. This procedure continues until the execution time is increased above an acceptable level, or a cost reduction is achieved by eliminating N pages of storage.

The proof of the slope selection optimization procedure requires that the problem statement be formalized slightly. Therefore, a series of definitions will be given, followed by the problem statement, followed by the optimization procedure (slope selection algorithm) and the proof that the slope selection algorithm performs as claimed.

The slope selection algorithm forms the basis of the optimization procedure of Figure 4.3.6. The procedure works on the fastest implementation set $(I^1)$ for a hardware option point. It determines

the implementation point which reduces the control storage $(S(I^1))$ by $S_{req} + \Delta S_{req}$ words while minimizing the increase in execution time. The slope selection algorithm is optimal only for a savings in storage of $S_{req} + \Delta S_{req}$ The procedure generates a new implementation point by replacement of the fastest implementation for a program type by an implementation which reduces the control storage requirements. The $\Delta S_{req}$ is the number of control words which are saved in excess of $S_{req}$ by the last implementation replacement made by the algorithm. Thus, $\Delta S_{req}$ will depend upon the particular H options, implementations, etc. The last step of the slope selection method flow chart of Figure 4.3.6 is to use a mixture of the fastest implementation method and its replacement implementation method to more closely equal $S_{req}$. The proof that follows ignores that extension and only proves the optimality of the slope selection algorithm.

Def. 4.3.1 $a_{ij}$ $a_{ij}$ is the jth implementation for the ith program type. The hardware options required by $a_{ij}$ are specified by $B_j^i$, the performance by $V_j^i$, and the required storage by $S_j^i$. Note that one $a_{ij}$ for each i requires no hardware options.

Def. 4.3.2 $D_j$ $D_j$ is the total number of implementations for the jth program type.

Def. 4.3.3 $\alpha$ $\alpha$ is the set of program type implementations.

$$\alpha = \{a_{ij} | p_i \in P,\ j \leq D_i\}$$

Def. 4.3.4 $\beta$ $\beta$ is a set of program type implementations such that there is at least one $a_{ij} \in \beta$ for each $p_i$.

$$\beta \subset \alpha$$

$$\forall \text{ one } a_{ij} \in \beta \text{ for each } p_i \in P$$

Def. 4.3.5 $\beta^i$ The $\beta^i$ set is a subset of the $\beta$ set with each element of $\beta^i$ an implementation method for the ith program type.

$$\beta^i = \{a_{ij} | a_{ij} \in \beta\}$$

Def. 4.3.6 I set An I set is a subset of $\beta$ with one and only one element from each $\beta^j$ set. The I set satisfies the following conditions.

$$|I \cap \beta^j| = 1 \quad \text{for all } j$$

$$I \subset \beta$$

Equations 4.3.1 and 4.3.2 are used to compute the difference in execution time and the difference in control storage if one program type implementation $(a_{ij})$ is replaced by another program type

implementation ($a_{ik}$). Equations 4.3.1 and 4.3.2 assume the existence of the frequency of occurrence vector O, the frequency of execution vector F, and the subroutine selection vector E All three vectors are defined in Section 3.2.

Equation 4.3.1

$$\Delta t^{i}_{jk} = F_i * O_i * (V^{i}_{k} - V^{i}_{j})$$

Equation 4.3.2

$$\Delta s^{i}_{jk} = [O_i * \overline{E}_i + E_i] * (S^{i}_{j} - S^{i}_{k})$$

Def. 4.3.7 Difference Vector (i, j, k, $\Delta t$, $\Delta s$, $\Delta t/\Delta s$):

The difference vector is a set of values computed on $\beta^i$, between $a_{ij}(a_{ij} \epsilon \beta^i)$ and the other elements of $\beta^i$. The algorithm for computing k, $\Delta t$, $\Delta s$, and $\Delta t/\Delta s$ is given below. The algorithm computes that element of $\beta^i$ (call it $a_{ik}$) which has the smallest $\Delta t/\Delta s$ slope relative to j or if none exists k is set to -1.

Algorithm

Step 1 compute $\Delta s^{i}_{j\ell}$ for each $a_{i\ell} \epsilon \beta^i$

Step 2 eliminate any $a_{i\ell}$ for which $\Delta s^{i}_{j\ell} \leq 0$

Step 3 compute $\Delta t^i_{j\ell}$ for the remaining elements.

Step 4 eliminate any $a_{i\ell}$ for which $\Delta t^i_{j\ell} \leq 0$.

Step 5 if there are no remaining elements, set $k = -1$ and go to Step 9.

Step 6 compute $\Delta t^i_{j\ell}/\Delta s^i_{j\ell}$ for the remaining elements.

Step 7 select the element from those remaining which has the smallest $\Delta t/\Delta s$. If more than one element has the same $\Delta t/\Delta s$ and if it is the smallest value select the one which has the largest $\Delta s$ value. Call that element $a_{i\ell}$.

Step 8 set $k = \ell$, $\Delta t = \Delta t^i_{j\ell}$, $\Delta s = \Delta s^i_{j\ell}$, and $\Delta t/\Delta s = \Delta t^i_{j\ell}/\Delta s^i_{j\ell}$.

Step 9 Stop.

Def. 4.3.8 $I^1$ set

The $I^1$ set is a subset of $\beta$ such that:

$$I^1 = \{a_{ij} | v^i_j \leq v^i_k \text{ all } a_{ik} \in \beta^i \text{ and if } v^i_j = v^i_k \quad s^i_j \leq s^i_k\}$$

Slope selection algorithm

$T_{max}$ is the maximum increase in execution time which can be accepted.

$S_{req}$ is the number of control storage words which must be saved.

Step 1 Set $i = 0$

Step 2 $i = i+1$

Step 3 if $i > |P|$ go to Step 10

Step 4 select $a_{ij} \epsilon I^1$

Step 5 compute the difference vector

$(i, j, k, \Delta t, \Delta s, \Delta t/\Delta s)$

Step 6 if $k = -1$ go to Step 2

Step 7 place $(i, j, k, \Delta t, \Delta s, \Delta t/\Delta s)$ into the queue $Q$

Step 8 set $j = k$

Step 9 go to 5

the preceding steps have calculated the 1st, 2nd, etc. difference for each $\beta^i$ set.

Step 10 order the elements of $Q$ with $Q^1$ the element $(i, j, k, \Delta t, \Delta s, \Delta t/\Delta s)$ having the smallest $\Delta p/\Delta s$

$Q^2$ the second smallest, etc.

$\therefore$ using the notation $\Delta t/\Delta s^1 = \Delta t/\Delta s$ of $Q^1$

$\Delta t^1 = \Delta t$ of $Q^1$

$$i^1 = i \text{ of } Q^1$$

$$\Delta t/\Delta s^1 < \Delta t/\Delta s^2 < \Delta t/\Delta s^3 \cdots$$

Step 11 copy set $I^1$ into set $I^2$

Step 12 $dT = 0$

Step 13 $ds = 0$

Step 14 $\ell = 1$

Step 15 $ds = ds + \Delta s^{\ell}$

Step 16 if $ds \geq S_{req}$ go to Step 23

Step 17 $dT = dT + \Delta t^{\ell}$

Step 18 if $dT > T_{max}$ go to Step 27

Step 19 $I^2 = I^2 - a_{i^{\ell}j^{\ell}} + a_{i^{\ell}k^{\ell}}$

Step 20 $\ell = \ell + 1$

Step 21 if $\ell > |Q|$ go to Step 27

Step 22 go to Step 15

Step 23 $dT = dT + \Delta t^{\ell}$

Step 24 if $dT > T_{max}$ go to Step 27

Step 25 $I^2 = I^2 - a_{i^{\ell}j^{\ell}} + a_{i^{\ell}k^{\ell}}$

Step 26 success stop

Step 27 stop it is not possible to save a unit page of storage

Def. 4.3.10 $I_N$ $I_N$ is the I set generated after the Nth iteration of Step 19 of the slope selection algorithm.

Def. 4.3.11 $S(I_N)$ The function $S(I_N)$ specifies the number of words of control storage required by the implementation set $I^N$

Def. 4.3.12 $T(I_N)$ The function $T(I_N)$ specifies the number of microcycles to execute a complete sequence of program types (the number of executions is specified by the F vector and the 0 vector).

Def. 4.3.13 $\Delta T(I_N)$ $\Delta T(I_N) = T(I_N) - T(I^1)$

Def. 4.3.14 $\Delta S(I_N)$ $\Delta S(I_N) = S(I^1) - S(I_N)$

Lemma 4.3.3 if $\Delta S(I_N) \leq \Delta S(I')$ $I' \subset \beta$, $I_N \subset \beta$

then $\Delta T(I_N) \leq \Delta T(I')$

Proof: $I_N$ and $I'$ are constructed from $I^1$ by replacing implementation methods of $I^1$ by implementation methods which reduce $S(I^1)$

$\therefore$ any of the replacement implementations are represented in the order queue Q produced by the slope selection algorithm.

Let $\lambda$ be the set of elements of Q ($Q^i$, $Q^2$, etc.) which would be used to construct $I'$.

$$\therefore \quad \Delta T(I') = \sum_{\lambda} \Delta t^i$$

$$\Delta S(I') = \sum_{\lambda} \Delta s^i$$

let $\lambda_N$ be the first N elements of Q

$(Q^1, Q^2, \ldots, Q^N)$

let $\lambda_a = \lambda_N \cap \lambda$

let $\lambda_b = \lambda_N - \lambda_a$

let $\lambda_c = \lambda - \lambda_a$

let $t_a = \sum_{\lambda_a} \Delta t^i$

$$s_a = \sum_{\lambda_a} \Delta s^i$$

$$t_b = \sum_{\lambda_b} \Delta t^i$$

$$s_a = \sum_{\lambda_b} \Delta s^i$$

$$\therefore \quad \Delta t(I_N) = t_a + t_b$$

$$\Delta s(I_N) = s_a + s_b$$

$$t_c = \sum_{\lambda_c} \Delta t^i$$

$$s_c = \sum_{\lambda_c} \Delta s^i$$

$\therefore \quad \Delta T(I') = t_a + t_c$

$\Delta S(I') = s_a + s_c$

1) $\frac{t_a}{s_a} \leq \frac{\Delta t^N}{\Delta s^N}$ since $\lambda_a \subset \lambda_N$

2) $\frac{t_b}{s_b} \leq \frac{\Delta t^N}{\Delta s^N}$ since $\lambda_b \subset \lambda_N$

3) $\frac{t_c}{s_c} \geq \frac{\Delta t^N}{\Delta s^N}$ since $\lambda_c \cap \lambda_N = \Phi$

$\therefore$ if $\Delta S(I_N) \leq \Delta S(I')$

$s_a + s_b \leq s_a + s_c$

$\therefore \quad s_b \leq s_c$

since $\frac{t_b}{s_b} \leq \frac{t_c}{s_c}$ by 2 and 3

$\frac{t_b}{s_c} \leq \frac{t_c}{s_c}$

$\therefore \quad t_b \leq t_c$

$\therefore \quad t_a + t_b \leq t_a + t_c$

$\therefore \quad T(I_N) \leq T(I')$

Theorem 4.3.3 Given a $\beta$ set and values $S_{req}$ and $P_{max}$, the slope selection algorithm will generate an I set $(I^2)$ if one exists such that

1) $T(I^2) - T(I^1) \leq T_{max}$

2) $S(I^2) - S(I^2) = S' \geq S_{req}$

3) No $I^3$ $(I^3 \neq I^2)$ exists such that if

$$S(I^1) - S(I^3) \geq S'$$

$$T(I^3) - T(I^1) < T(I^2) - T(I^1)$$

Proof: 1) Condition 1 is a direct result of Step 17 of the algorithm.

2) Condition 2 is a direct result of Step 16 of the algorithm.

3) Condition 3 is a direct result of Lemma 4.3.3.

# Chapter V

## HARDWARE SELECTION EXAMPLE

In this chapter a complete design example will be given. The example begins with the presentation of a base architecture, a set of hardware options, and their associated cost functions. Next the implementation methods for a selected set of program types are given. Finally, the optimization program is applied to the data developed in the first part of the chapter. As a demonstration of the flexibility of the program, cost performance curves will be developed for different values of control storage bit cost, control storage page size, and frequency of execution.

## 5.1 The Design Method

The design procedure is outlined below.

Step 1. A base set of hardware is provided.

Step 2. A set of hardware options and their associated cost functions are provided.

Step 3. A set of implementations for each PTL program type is developed. One implementation requires only the base hardware while any additional program type implementations use one or more hardware options to either improve the performance or reduce the storage requirement.

Step 4. Statistics on a set of PTL programs are gathered. The statistics specify the expected frequency of occurrence and the expected frequency of execution of each program type.

Step 5. The optimization program is used to develop the optimal cost performance points. The input to the optimization program is a value for required performance and the information developed in Steps 1 through 4. The output from the optimization programs specifies the least costly set of hardware options and the implementations which will achieve the desired performance goals.

## 5.2 Base Hardware and Hardware Options

A block diagram of the base hardware is shown in Figure 5.2.1. The main components of the base hardware are the microprogram control unit, the core memory, the memory address register, the memory buffer register, the high speed scratch pad memory, the arithmetic logic unit, and their associated data paths. The features of each of the base hardware units are summarized in Table 5.2.1.

The base hardware set requires a 20 bit microinstruction word length. There are four basic word forms which are named the MEM, ALU, BRANCH, and I/O-EMIT words. The different instruction formats and their associated microorders are given in Table 5.2.2. Examples of some possible microinstructions are given in Table 5.2.3. Some special considerations in using the base microinstructions are listed in Table 5.2.4.

A block diagram of the base hardware combined with the hardware options is shown in Figure 5.2.2. The hardware options include features which provide additional testing options, shifting options, floating point operations. The features of each of the hardware options are summarized in Table 5.2.5. The cost function for each hardware option is given in Table 5.2.6. The cost of the hardware options is interdependent and this fact is reflected in the cost functions of Table 5.2.6. The additional microbits required for each

hardware option are listed in Table 5. 2. 7. An explanation of the values listed in Tables 5. 2. 6 and 5. 2. 7 will be given below on a hardware option by hardware option basis.

1) BR =: The cost of hardware option 1 depends upon the inclusion or the exclusion of options 4, 5, and 7. The effects of including either or both options 4 and 7 with option 1 can be considered in the cost functions of options 4 or 7. Option 5 specifies the addition of a second input bus. Therefore, if option 5 is added, extra circuitry will be required to select among two possible input buses and the cost of option 1 increases. The additional microbits required by option 1 are used to specify the addresses of data and to control the option. If option 5 is specified an additional microbit is required to select among the input buses.

2) ALU TEST: The cost of hardware option 2 depends upon the inclusion or exclusion of hardware option 8, which is the 16 bit ALU. If a 16 bit ALU is chosen instead of the 8 bit ALU, the cost of the test circuitry will increase.

There are no additional microorders required if hardware option 8 is chosen, since no additional tests are performed and no new data need be specified.

3) COUNT-1: The cost of hardware option 3 depends upon the inclusion or exclusion of hardware option 5. Option 5 specifies the addition of a second input bus. Therefore, if option 5 is added, extra circuitry will be required to select among the two possible input buses. The added microbits required by option 3 are used to input data to the device and to control the counting. If option 3 and option 5 are both specified an additional microorder is required to specify which input bus contains the data. However, because of field encoding no additional microbits are required.

4) BR ><: The cost of hardware option 4 depends upon the inclusion or exclusion of options 1, 5, and 7. However, only the cost of options 1 and 5 need be considered since the effects of including option 7 can be considered in the cost function of option 7. If option 5 is specified, but not option 1, then the cost

of option 4 is increased to cover the added circuitry to select among the two possible (option 5 specifies an extra input bus) input buses. If option 1 is specified then the cost of option 5, if specified, will have been covered by the cost function for option 1. However, it is assumed that options 1 and 4 could be combined into a single package. Therefore, if option 1 is specified the cost of option 4 is less since part of the cost has previously been absorbed in the cost of option 1. If option 1 is included with option 4 the microbits required to address data have already been assigned by option 1, otherwise 8 microbits for addressing are required. If option 5 is specified but option 1 is not specified an additional microbit for data bus selection is required.

5) Input Bus: The cost of hardware option 5 depends upon the inclusion or exclusion of option 7. The cost of selecting between the two buses is absorbed by the other hardware options. However, hardware option 7 specifies that the BR =, and BR >< are to be a full word test. Therefore, if option 7 is specified the width of the input bus must be doubled. If option 5 is

specified additional microbits are required to specify the addresses of data. Since the additional data bus is used for word formats, ALU (S2) and BRANCH (S3), the microbits must be added to both S2 and S3.

6) Shift: The cost of hardware option 6 depends upon the inclusion or exclusion of option 8. The shift unit will operate upon 16 bit words instead of 8 bit words if option 8 (full word ALU) is used. Thus the cost of option 6 will increase if option 8 is also specified. The additional microbits required by option 6 are used to select the type of shift and to specify the number of places to shift. If option 8 is specified one extra microbit is required to allow the specification of the longer shift.

7) Double Byte: The costs of options 1, 4, 5, and 7 are related. The cost of options 1 and/or 4 increases if option 7 is specified since the number of bits to be tested doubles. If option 5 is specified the cost of selecting the input data bus must be included for the additional 8 bits. The cost resulting from option 5 for a half word (8 bits) has been included in the

cost of options 1 and 4. Thus, the cost for option 7 depends upon options 1, 4, 5, and for each combination is the difference between the actual cost and the cost specified for option 1 and 4.

The inclusion of option 7 does not cause a requirement for any added microorders.

8) ALU: The cost of hardware option 8 is independent of the inclusion or exclusion of any other hardware options. The cost of the ALU is the difference between the cost of the standard 8 bit ALU and the optional 16 bit ALU. Thus the cost of the hardware option 8 is the exchange cost. No additional microbits are required to control the order ALU.

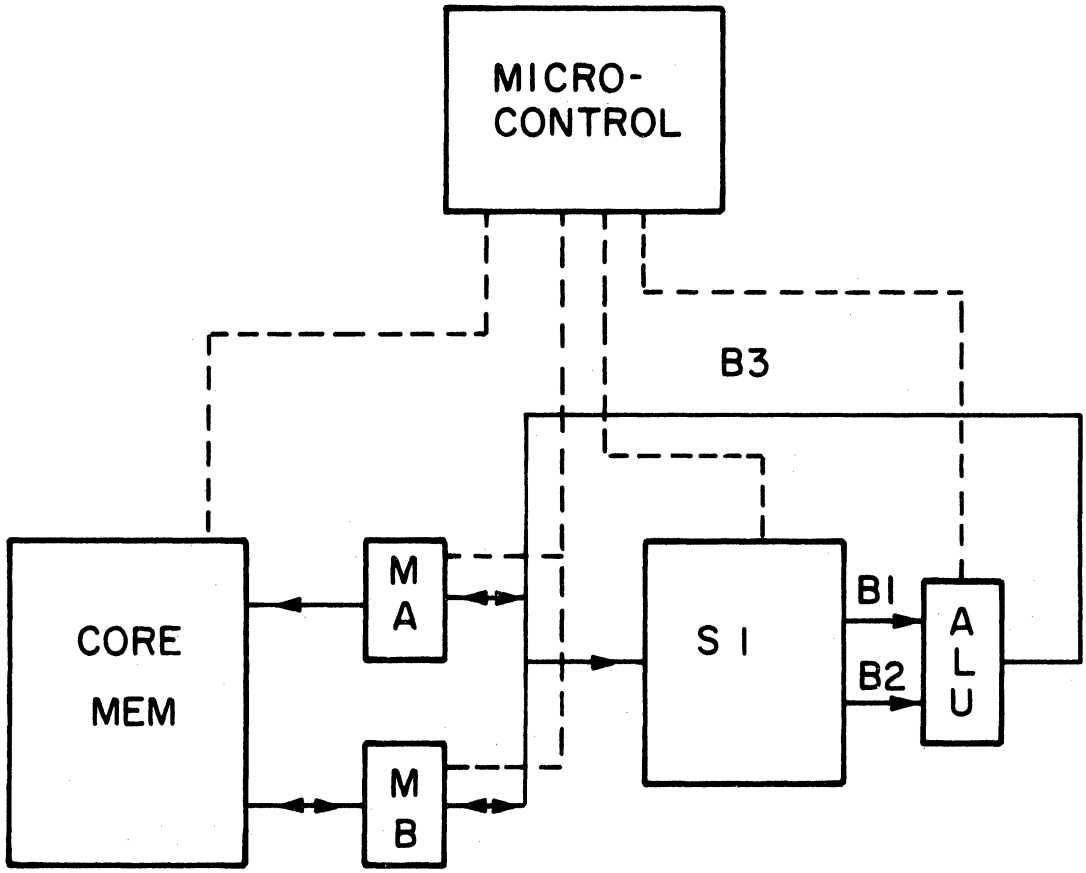
9) STACK: The cost of hardware option 9 is increased if hardware option 5 is specified, since additional circuitry is required to select the data bus. The additional microbits are required to control the input to and the output from the stack. If option 5 is specified an additional microbit is required to select the proper input buses.

10) Float A/S: The cost of hardware option 10 is independent of the inclusion or exclusion of any other hardware option.

To simplify the presentation of the data an exclusion relationship has been applied. This relationship prohibits hardware points which contain certain combinations of hardware options. For example the second input bus (option 5) is not allowed if at least one of options 1, 3, 4, or 9 is not included. Thus, if the second bus (option 5) is not connected to anything it should not be allowed. Such a situation would automatically be eliminated by the optimization program but its inclusion allows the optimization to be speeded up. The other exclusion relation specifies that option 7 (double byte) is not allowed if both option 1 and 4 are not included. Since option 7 only applies to 1 and 4, such a restriction is necessary. The two exclusion relations are listed below.

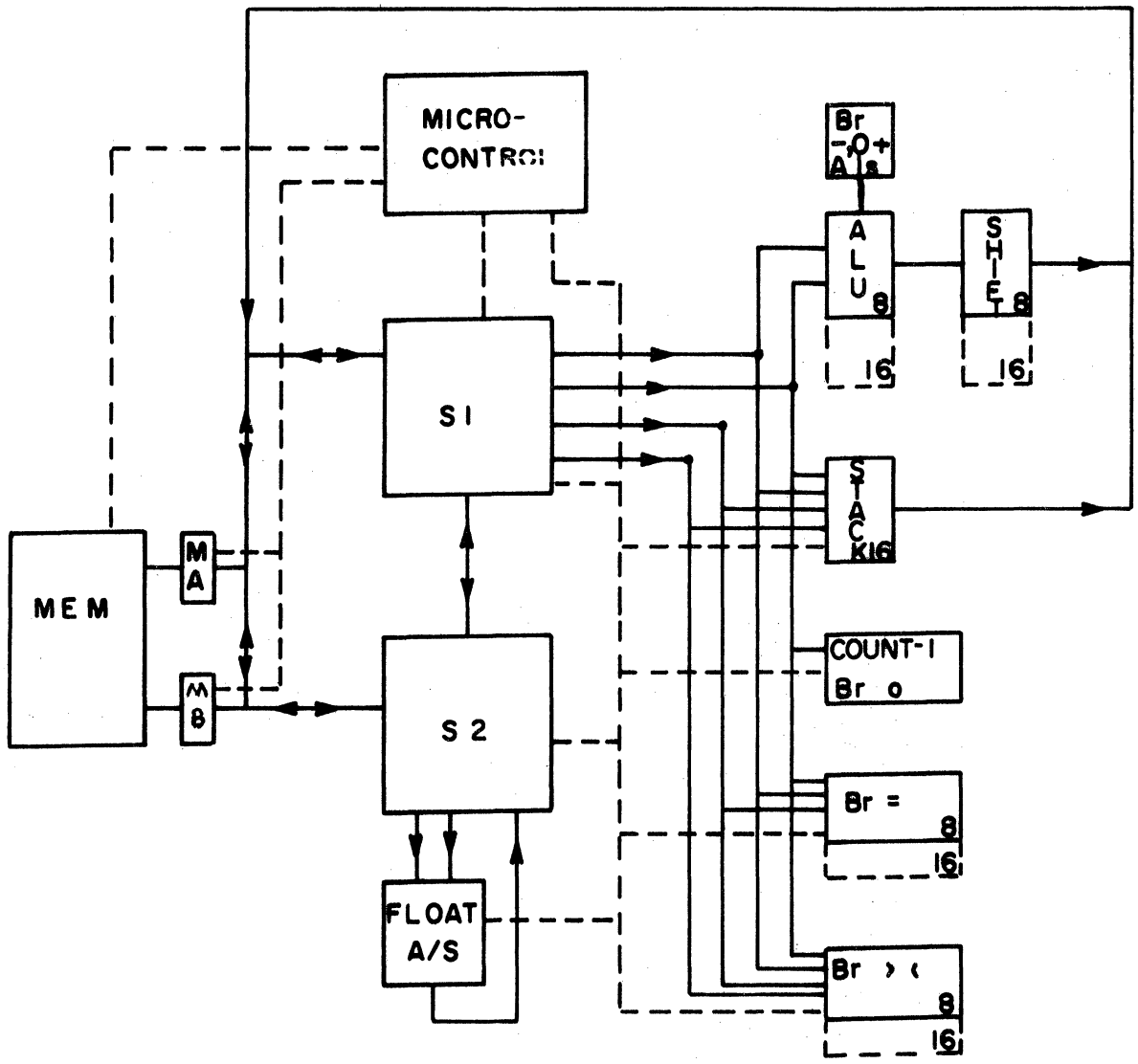
1) $\overline{1}, \overline{3}, \overline{4}, 5, \overline{9}$

2) $\overline{1}, \overline{4}, 7$




BASE HARDWARE CONFIGURATION

Figure 5.2.1




Base Hardware Plus Hardware Options

Figure 5.2.2

MICROCONTROL UNIT: The microcontrol unit controls the ALU, controls the gatting of data within the CPU, controls the gatting of data between the CPU and memory, and controls the memory. The microcontrol instructions are executed sequentially, but conditional branch tests are provided which may alter the execution sequence.

MEM: The core memory contains up to 32 K by 16 bit words and is addressable by words. The memory can be read or written in 3 microcycle time units.

MA: The MA register is 16 bits wide and specifies the location of core memory to be read or written.

MB: The MB is 16 bits wide and contains the contents of core (read) or the data to be written.

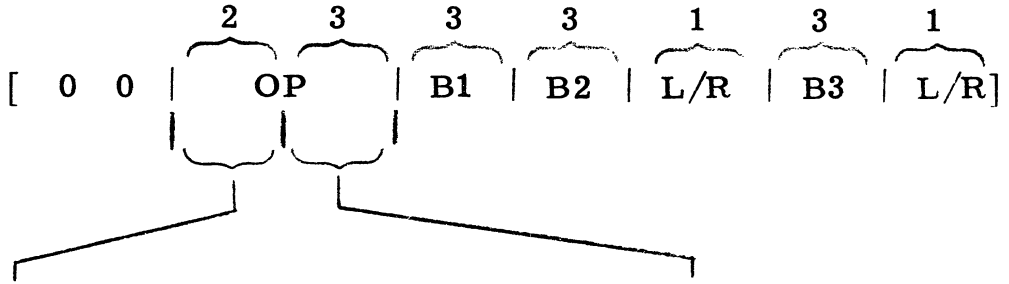
S1: The high speed scratch pad memory contains 64 by 16 bit words. The memory can be read and written in one microcycle time unit. The memory is addressable in a section by halfwords with 8 words per section. The section address is held in a microcontrol register and is set under microcontrol.

Base Hardware
Table 5.2.1

| | |
|---|---|
| ALU: | The ALU unit operates on 8 bit words (1/2 words) from S1. The unit can perform AND, OR, EOR, ADD, ADD+1, ADD + CARRY, SHIFT LEFT 1, SHIFT LEFT 1 WITH CARRY, SHIFT RIGHT 1, SHIFT RIGHT 1 WITH CARRY, DECIMAL ADD, DECIMAL ADD WITH CARRY, DECIMAL SUBTRACT, and DECIMAL SUBTRACT WITH CARRY. The carry is set by the arithmetic and shift operations and is unchanged until the next arithmetic or shift operation. The contents of the B1 bus may be complimented before being passed to the ALU input. The carry, and 8 ALU output bits are latched and can be tested under microcontrol. |
| B1, B2: | The data paths are 8 bits wide. |
| B3: | This data path is 16 bits wide. |

Base Hardware
Table 5.2.1 (Cont.)

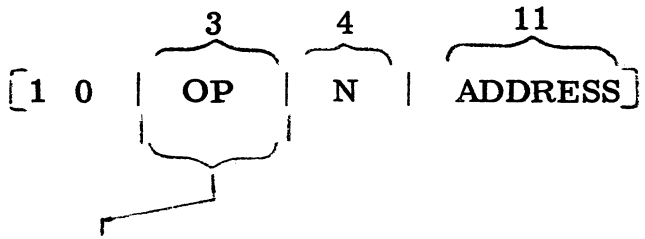
MEM




| | | | | | |
|---|---|---|---|---|---|
| 0 | 0 | MEMORY READ | MEMR | MA → S1 (B1 \|B2 \|B3) | MAO |
| 0 | 1 | MEMORY WRITE | MEMW | MB → S1 (B1 \|B2 \|B3) | MBO |
| 1 | 0 | MEMORY CYCLE | MEMC | MB → MA | MBMA |
| 1 | 1 | NO MEMORY OPERATION | | MA → MB | MAMB |
| | | | | S1 (B3 \|[ B1(L/R) \|B2(L/R)]) → MA | MAI |
| | | | | S1 (B3 \|[ B1(L/R) \|B2(L/R)]) → MB | MBI |
| | | | | NO OPERATION | NOP |
| | | | | O → B3 | B3←'O' |

Base Hardware Instruction Format
Table 5. 2. 2

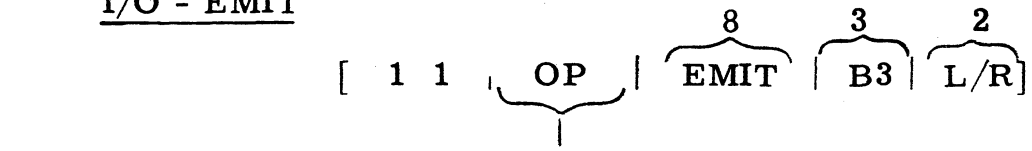
**BRANCH**




| | | |
|---|---|---|
| BRANCH IF CARRY SET | BRC | |
| BRANCH IF CARRY NOT SET | BRNC | |
| BRANCH IF ALU(LAST) $Bit_N = 0$ | BRAO | N |
| BRANCH IF ALU(LAST) $Bit_N = 1$ | BRA1 | N |
| BRANCH IF MB $Bit_N = 0$ | BRMBO | N |
| BRANCH IF MB $Bit_N = 1$ | BRMB1 | N |
| BRANCH Unconditional | BR | |
| BRANCH IF CA $\neq$ A(8) | BROF | |

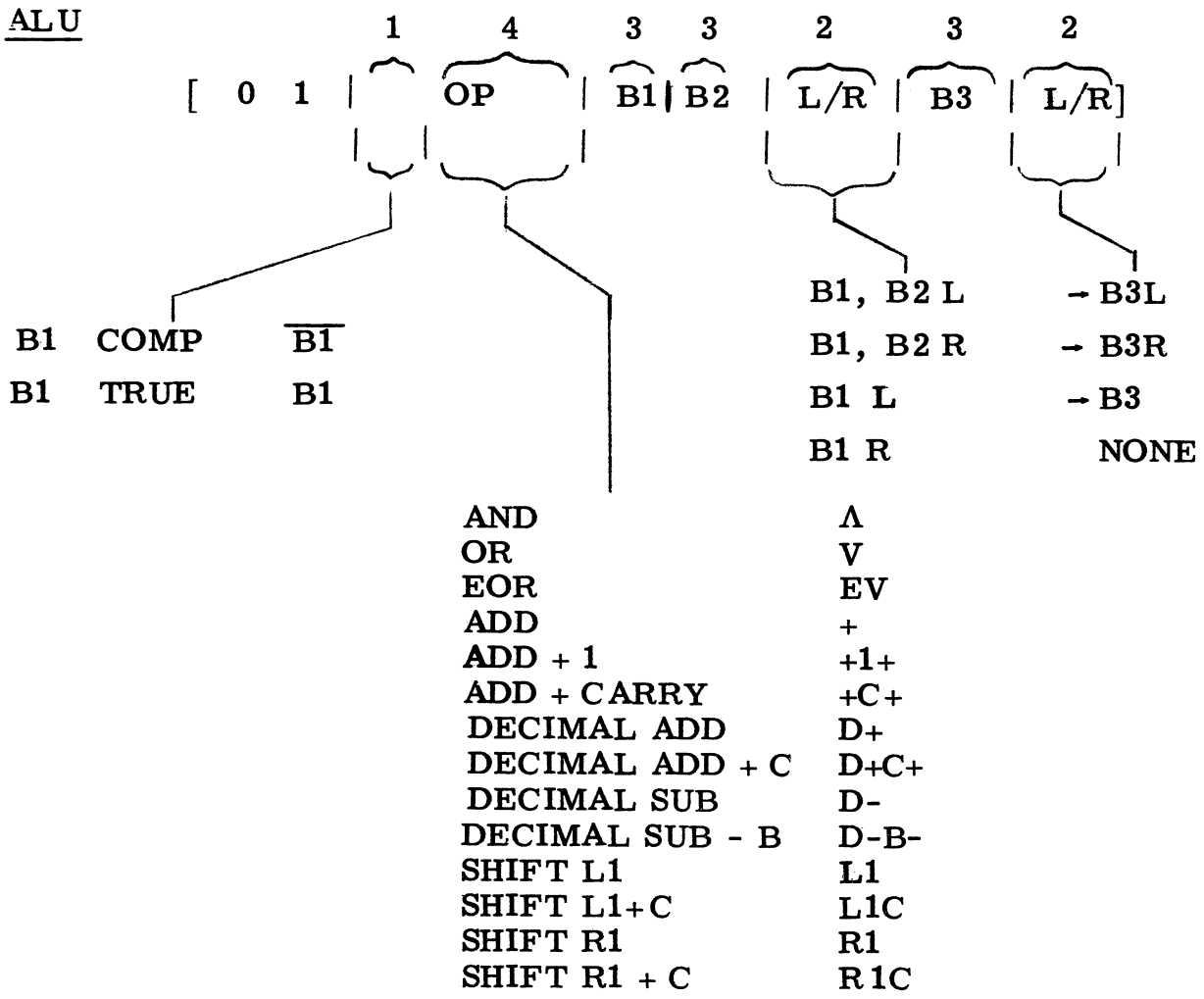
Base Hardware Instruction Format (Cont.)
Table 5.2.2

I/O - EMIT




| | |
|---|---|
| I/O D ← B3(L/R) | I/ODO |
| I/O D ← B3(L/R) | I/ODI |
| I/O S ← B3(L/R) | I/OSI |
| EMIT ← I/O D | ' ' ← I/OD |
| EMIT ← I/O S | ' ' ← I/OS |
| EMIT ← B3 L/R | ' ' ← B3 |
| EMIT ∧ B2 LR ← B2 L/R | ' ' ∧ B2(X) |
| EMIT ∨ B2 LR ← B2 L/R | ' ' ∨ B2(X) |
| EMIT + B2 LR ← B2 L/R | ' ' + B2(X) |
| EMIT+C +B2 LR ← B2 L/R | ' ' + C + B2(X) |
| EMIT .EV. B2 LR ← B2 L/R | ' ' .EV. B2(X) |
| SET S2 ADDRESS | SET S2 |
| EMIT ← MAL | ' ' ← MAL |
| EMIT ← MAR | ' ' ← MAR |
| EMIT ← MBL | ' ' ← MBL |
| EMIT ← MBR | ' ' ← MBR |

Base Hardware Instruction Format (Cont.)
Table 5. 2. 2




Base Hardware Instruction Format (Cont.)
Table 5. 2. 2.

Transfer S1 location A to MA and start a memory read.

MA ← B(A) MEMR

Transfer MA to MB and start a memory cycle.

MB ← MA MEMC

Add S1 location A to S1 location B and store in S1 location C.

RB3(C) ← RB2(A) + RB1(B)

LB3(C) ← RB2(A) + C + RB1(B)

Complement S1 location A and store in S1 location A.

RB3(A) ← $\overline{\text{RB1}}$(A)

LB3(A) ← $\overline{\text{LB1}}$(A)

Branch to L1 if S1 location A has a 1 in bit 2.

← RB1(A)

BRA1 2 L1

Branch to L1 if S1 location A has a 0 in bit 9

← LB1(A)

BRA0 1 L1

Using the Emit field zero the odd bit position of S1 location A

RB3(A) ← RB2(A) ∧ ' $AA_{16}$'

LB3(A) ← LB2(A) ∧ ' $AA_{16}$'

Microinstruction Examples
Table 5.2.3

i) The address specified in an ALU operation or an emit operation must be in the current S1 section.

ii) The B1 and B2 bus specification for ALU operations must specify the same half of their respective addresses. The B3 bus half may be different from the B1, B2 bus half.

| | |
|---|---|
| INCORRECT | RB3(C) ← RB1(A) + LB2(B) |
| CORRECT | RB3(C) ← RB1(A) + RB2(B) |
| CORRECT | RB3(C) ← LB1(A) + LB2(B) |

iii) The transfer of a full word from S1 to MA or MB; or from MA or MB to S1 for any S1 address may be accomplished with a MEM microinstruction as shown below.

MA ← B(A)

v) The value of the result of the last ALU operation may be tested using the BR microinstructions as shown below. N specifies the bit position to test.

BRA1 N 1

Base Hardware Programming Considerations
Table 5.2.4

1. BR=: The branch on equal option tests two words of S1 (8 bit or 16 bit dependent upon option 7). If the words are equal, a branch in the microprogramming occurs.

2. ALU TEST: The result of an ALU operation is tested for negative, zero, positive, or all one's. The test result is stored and may be used in a branch test after the completion of the current microcycle. The test status is stored until the next test microorder is issued.

3. COUNT-1: The option is used for microloops. A number is stored into the device and counted down under microcontrol. A microbranch is caused when the count goes to 0.

4. BR><: The branch >< option tests two words of S1(8 bits or 16 bits depending upon option 7). A branch in the microprogram occurs if the test is satisfied.

5. INPUT BUS: A second dual bus is provided from S1 to the hardware option functional units.

Summary of Hardware Options
Table 5.2.5

| | | |
|---|---|---|
| 6. | SHIFT: | The output from the ALU is shifted before placing the results on the output bus. The shift unit will be either an 8 bit or a 16 bit shift unit depending upon the choice of ALU's (option 8). The unit performs either a left or a right N place shift (end around, carry out, or arithmetic) under microcontrol. |
| 7. | Double Byte: | This option specifies whether options 1 and 4 will be an 8 bit (half word) or 16 bit (full word) test. |
| 8. | ALU: | The 8 bit base hardware ALU unit is replaced by a 16 bit ALU. Both ALU's perform the same arithmetic logic functions. |
| 9. | STACK: | This is a full word (16 bit) 1024 word stack. An item may be placed into or removed from the stack in 1 microcycle time unit. |
| 10. | Float A/S: | This unit performs a floating add or subtract operation, in one microcycle time unit, between any two double words (32 bit) of scratch pad memory S2. The scratch pad memory (S2) is part of the option. |

Summary of Hardware Options (Cont.)
Table 5.2.5

| Hardware Option Number | Cost | Constraint |
|---|---|---|
| 1 | 100.00 | $\overline{5}$ |
| | 110.00 | 5 |
| 2 | 150.00 | $\overline{8}$ |
| | 200.00 | 8 |
| 3 | 200.00 | $\overline{5}$ |
| | 210.00 | 5 |
| 4 | 210.000 | 1 |
| | 250.00 | $\overline{1}, \overline{5}$ |
| | 260.00 | $\overline{1}, 5$ |
| 5 | 300.00 | $\overline{7}$ |
| | 550.00 | 7 |
| 6 | 350.00 | $\overline{8}$ |
| | 600.00 | 8 |
| 7 | 200.00 | $\overline{1}, 4, \overline{5}$ |
| | 100.00 | $1, \overline{4}, \overline{5}$ |
| | 230.00 | $1, 4, \overline{5}$ |
| | 210.00 | $\overline{1}, 4, 5$ |
| | 110.00 | $1, \overline{4}, 5$ |
| | 250.00 | $1, 4, 5$ |
| 8 | 600.00 | |
| 9 | 700.00 | $\overline{5}$ |
| | 710.00 | 5 |
| 10 | 1200.00 | |

Hardware Option Costs
Table 5. 2. 6

| Hardware Option Number | Microorders | Microbits | S word | Constraint |
|---|---|---|---|---|
| 1 | 1 | 9 | S3 | $\overline{5}$ |
| | 2 | 10 | S3 | 5 |
| 2 | 3 | 2 | S2 | |
| 3 | 2 | 2 | S2 | $\overline{5}$ |
| | 3 | 2 | S2 | 5 |
| 4 | 2 | 2 | S3 | 1 |
| | 2 | 12 | S3 | $\overline{1}, \overline{5}$ |
| | 3 | 13 | S3 | $\overline{1}, 5$ |
| 5 | 0 | 8 | S2, S3 | |
| 6 | 5 | 6 | S2 | $\overline{8}$ |
| | 5 | 7 | S2 | 8 |
| 7 | 0 | 0 | | |
| 8 | 0 | 0 | | |
| 9 | 2 | 2 | S2, S3, S4 | |
| | 3 | 3 | S2, S3, S4 | |
| 10 | 5 | 8 | S2, S3 | |

Added Microbits
Table 5. 2. 7

## 5.3 Cost Performance Curves

Using the base hardware and the hardware options described in Section 5.2, a set of implementation methods was developed. The list of program types for which implementations were developed is given in Table 5.3.1

Typically, the full set of implementations is not developed. The designer develops the implementations for the base hardware. By using the set of base hardware implementations, it is possible to specify the changes in storage requirements and execution time over that of the base implementation, which will occur if different hardware options are applied to an individual program type. That approach was used to develop the implementation values of Table 5.3.1. The base implementations are listed in Appendix B.

In this section a number of design studies will be performed using the optimization program. The design studies will investigate the effects upon the optimal cost performance curve which occur when various parameters are varied. The parameters which will be studied are control storage bit cost, control storage page size, and program type frequency of execution.

The cost performance curves will be developed using the optimization program described in Chapter IV and the values for the

hardware option costs and constraints of Table 5.2.6, the added microbits and constraints of Table 5.2.7, the implementation time storage and hardware options of Table 5.3.2, the frequency of execution and frequency of occurrence values of Table 5.3.3, and the base hardware information of Table 5.3.4. The performance plotted on each graph is the average execution time (Equation 3.2.11) and the cost is the cost of the base hardware, hardware options and control storage.

The cost performance studies which are presented below are broken up into several parts. The explanation of each set of curves will be given before the curves are presented.

1) S3 ← S1 ∧ S2

2) S3 ← S1  S2

3) S3 ← ~ S1

4) S1 ← S2 TR ADDR S3 LABEL

5) S1 ← SHIFT S2 CARRY OUT LEFT S3

6) S1 ← S1 + S2 INTEGER

7) S1 ← S1 - S2 INTEGER

8) S1 INTEGER TO FLOAT S2

9) S1 ← STACK LABEL

10) STACK ← S1 LABEL

11) CLEAR STACK

12) INCREMENT POINTER LINK LIST S1 LABEL

13) DECREMENT POINTER LINK LIST S1 LABEL

14) SET LINK LIST S1

15) EMPTY LINK LIST S1

16) REMOVE S1 LINK LIST S2 LABEL

17) INCR S1

18) DECR S1

19) DO LABEL S1

20) BR S1 = S2 LABEL

21) BR S1 < S2 LABEL

Program Types Used in Examples
Table 5.3.1

| Program Type | Implementation Method | Time (Microcycles) | Storage (Words) | Options |
|---|---|---|---|---|
| 1 | 1 | 2 | 2 | - |
| | 2 | 1 | 1 | 8 |
| 2 | 1 | 2 | 2 | - |
| | 2 | 1 | 1 | 8 |
| 3 | 1 | 2 | 2 | - |
| | 2 | 1 | 1 | 8 |
| 4 | 1 | 1 | 9 | - |
| | 2 | 9 | 7 | 8 |
| | 3 | 9 | 7 | 4,7 |
| | 4 | 8 | 6 | 4,7,8 |
| | 5 | 7 | 5 | 4,5,7,8 |
| 5 | 1 | 17 | 26 | - |
| | 2 | 15 | 6 | 3 |
| | 3 | 13 | 18 | 8 |
| | 4 | 10 | 5 | 3,8 |
| | 5 | 1 | 1 | 6 |
| 6 | 1 | 4 | 5 | - |
| | 2 | 3 | 4 | 8 |
| | 3 | 2 | 2 | 2 |
| | 4 | 1 | 1 | 2,8 |

Time, Storage, and Hardware Option Requirements of Different Implementation Methods for Program Types of Table 5.3.1

Table 5.3.2

| Program Type | Implementation Method | Time (Microcycles) | Storage (Words) | Options |
|---|---|---|---|---|
| 7 | 1 | 4 | 5 | - |
| | 2 | 3 | 4 | 8 |
| | 3 | 2 | 2 | 2 |
| | 4 | 1 | 1 | 2,8 |
| 8 | 1 | 6.5 | 10 | - |
| | 2 | 5.5 | 8 | 8 |
| | 3 | 1 | 1 | 10 |
| 9 | 1 | 8 | 10 | - |
| | 2 | 6 | 8 | 8 |
| | 3 | 1 | 1 | 9 |
| 10 | 1 | 7 | 12 | - |
| | 2 | 5 | 8 | 8 |
| | 3 | 1 | 1 | 9 |
| 11 | 1 | 3 | 3 | - |
| | 2 | 2 | 2 | 8 |
| | 3 | 1 | 1 | 9 |
| 12 | 1 | 6 | 6 | - |
| 13 | 1 | 9 | 9 | - |
| | 2 | 7 | 7 | 8 |

Time, Storage, and Hardware Option Requirements of Different Implementation Methods for Program Types of Table 5.3.1

Table 5.3.2 (Continued)

| Program Type | Implementation Method | Time (Microcycles) | Storage (Words) | Option |
|---|---|---|---|---|
| 14 | 1 | 7 | 7 | - |
| 15 | 1 | 27 | 11 | - |
| 16 | 1 | 20 | 23 | - |
| | 2 | 17 | 20 | 8 |
| 17 | 1 | 4 | 5 | - |
| | 2 | 3 | 4 | 8 |
| | 3 | 2 | 2 | 2 |
| | 4 | 1 | 1 | 2,8 |
| 18 | 1 | 4 | 5 | - |
| | 2 | 3 | 4 | 8 |
| | 3 | 2 | 2 | 2 |
| | 4 | 1 | 1 | 2,8 |
| 19 | 1 | 4 | 4 | - |
| | 2 | 3 | 3 | 8 |
| | 3 | 1 | 2 | 3 |
| 20 | 1 | 4 | 6 | - |
| | 2 | 3 | 3 | 8 |
| | 3 | 2.5 | 3 | 1 |
| | 4 | 2 | 2 | 2,8 |
| | 5 | 1 | 1 | 1.7 |

Time, Storage, and Hardware Option Requirements of Different Implementation Methods for Program Types of able 5.3.1

Table 5.3.2 (Continued)

| Program Type | Implementation Methods | Time (Microcycles) | Storage (Words) | Option |
|---|---|---|---|---|
| 21 | 1 | 6 | 11 | - |
| | 2 | 5 | 10 | 8 |
| | 3 | 2.5 | 3 | 4 |
| | 4 | 1 | 1 | 4,7 |

Time, Storage, and Hardware Option Requirements of Different Implementation Methods for Program Types of Table 5.3.1

Table 5.3.2 (Continued)

| Program Type | Frequency Execution | Frequency Occurrence |
|---|---|---|
| 1 | 1.0 | 3.0 |
| 2 | 1.0 | 3.0 |
| 3 | 1.0 | 3.0 |
| 4 | 1.0 | 3.0 |
| 5 | 1.0 | 3.0 |
| 6 | 1.0 | 3.0 |
| 7 | 1.0 | 3.0 |
| 8 | 1.0 | 3.0 |
| 9 | 1.0 | 3.0 |
| 10 | 1.0 | 3.0 |
| 11 | 1.0 | 3.0 |
| 12 | 1.0 | 3.0 |
| 13 | 1.0 | 3.0 |
| 14 | 1.0 | 3.0 |
| 15 | 1.0 | 3.0 |
| 16 | 1.0 | 3.0 |
| 17 | 1.0 | 3.0 |
| 18 | 1.0 | 3.0 |
| 19 | 1.0 | 3.0 |
| 20 | 1.0 | 3.0 |
| 21 | 1.0 | 3.0 |

Frequency of Occurrence and Frequency of Execution of the Program Types of Table 5.3.1

Table 5.3.3

| | |
|---|---|
| Base Hardware Cost | $10,000.00 |
| Base Hardware Control Word Size | 20 bits |
| Control Storage Page Size | 128 words |
| Control Storage Bit Cost | $0.10 |

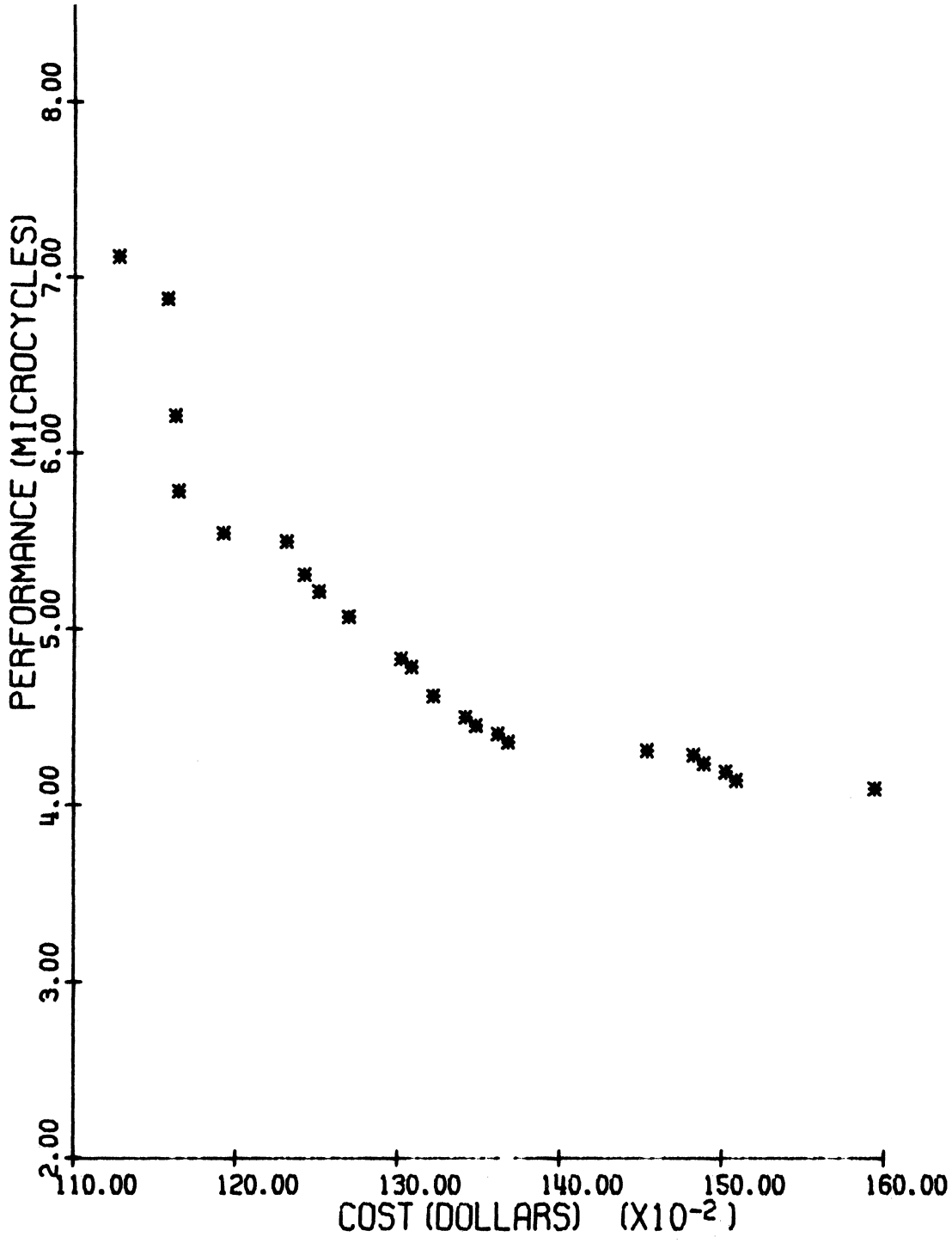
Base Hardware Values

Table 5.3.4

Standard Configuration

The first curve is for the standard configuration. The other cost performance curves are developed by varying one of the parameters used to generate the standard curve.

The curve of Figure 5.3.1 represents the optimum cost performance curve for the standard values (Table 5.2.6, 5.2.7, 5.3.2, 5.3.3, 5.3.4). The exact execution time and cost values and the required hardware options are listed in Table 5.3.5. The first point requires hardware option 2, the second hardware options 2, 3, the third hardware option 8, and the fourth point requires hardware options 2, 8. The difference in execution time between the point requiring options 2, 3 and the point requiring option 8 is .6 microcycles but the cost difference is only 50 dollars. This result occurs because option 8 is only slightly more expensive than options 2 and 3 combined but option 8 can be used to improve the performance of almost all of the program types. The costs of the point requiring option 8 and that requiring options 2 and 8 are almost identical. This result occurs because the combination of options 2 and 8 saves enough control storage words to almost offset the cost of combining option 2 with option 8.




Optimum Cost Performance Curve
Standard Configuration (Tables 5.2.6, 5.2.7, 5.3.2, 5.3.3, 5.3.4)

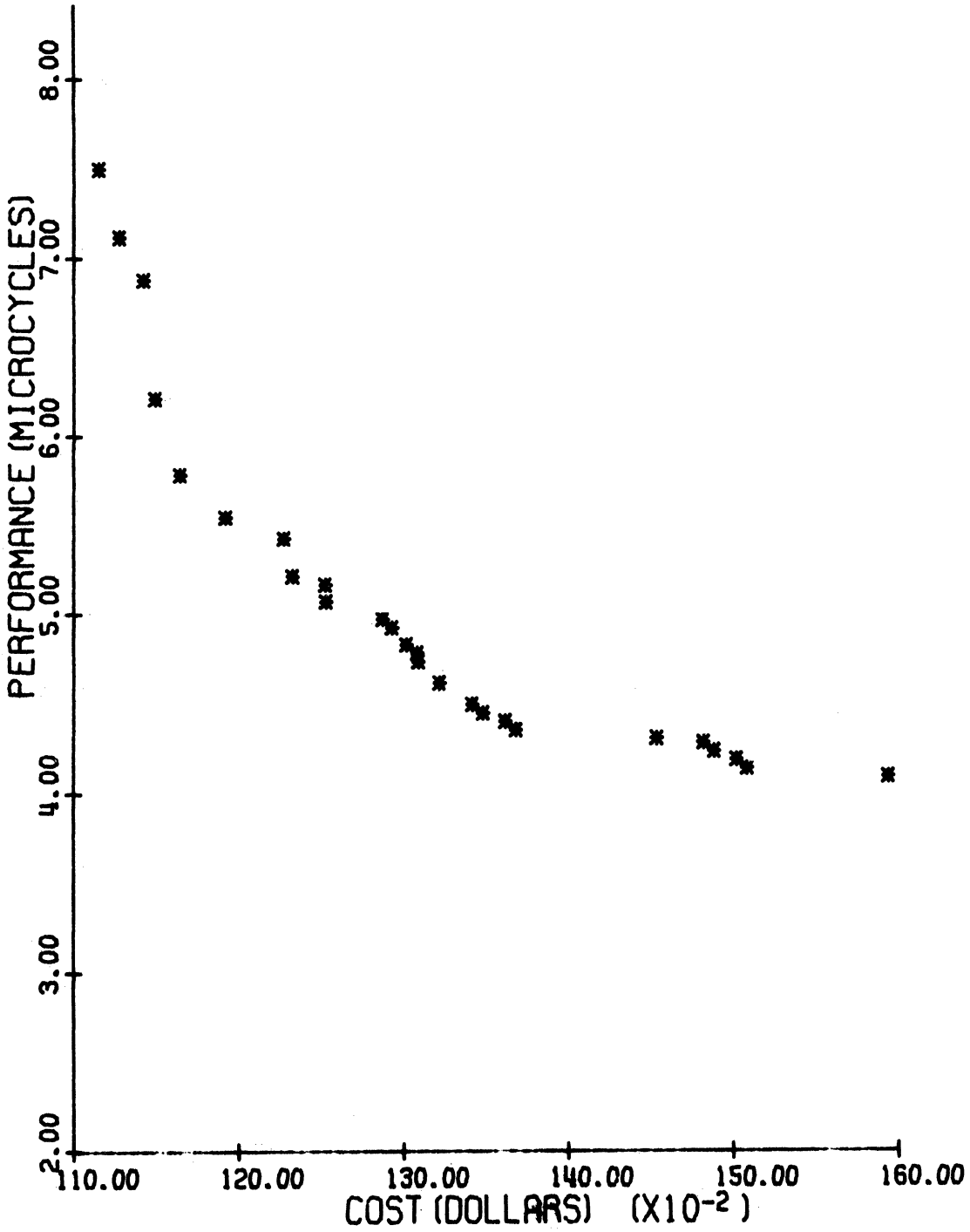
Figure 5.3.1

| PERFORMANCE (AVERAGE MICROCYCLES) | COST (DOLLARS) | HARDWARE OPTIONS |
|---|---|---|
| 7.119 | 11276.39 | 2 |
| 6.881 | 11578.80 | 2,3 |
| 6.214 | 11624.00 | 8 |
| 5.786 | 11644.80 | 2,8 |
| 5.548 | 11921.60 | 2,3,8 |
| 5.500 | 12313.60 | 1,2,3,7,8 |
| 5.309 | 12421.60 | 2,8,9 |
| 5.214 | 12513.60 | 2,6,8 |
| 5.071 | 12698.39 | 2,3,8,9 |
| 4.833 | 13020.39 | 2,3,4,7,8,9 |
| 4.786 | 13084.80 | 1,2,3,4,7,8,9 |
| 4.619 | 13220.39 | 2,4,6,8,9 |
| 4.500 | 13420.39 | 2,4,6,7,8,9 |
| 4.452 | 13484.80 | 1,2,4,6,7,8,9 |
| 4.405 | 13620.39 | 2,3,4,6,7,8,9 |
| 4.357 | 13684.80 | 1,2,3,4,6,7,8,9 |
| 4.309 | 14540.80 | 1,2,3,4,5,6,7,8,9 |
| 4.286 | 14825.20 | 2,4,6,7,8,9,10 |
| 4.238 | 14889.60 | 1,2,4,6,7,8,9,10 |
| 4.190 | 15025.20 | 2,3,4,6,7,8,9,10 |
| 4.143 | 15089.60 | 1,2,3,4,6,7,8,9,10 |
| 4.095 | 15945.60 | 1,2,3,4,5,6,7,8,9,10 |

Optimum Cost-Performance and Required Hardware Options for Standard Values (Tables 5.2.6, 5.2.7, 5.3.2, 5.3.3, 5.3.4)
Table 5.3.5

Page Size

The page size parameter was varied to generate the curves below. The curve of Figure 5.3.2 (Table 5.3.6) is for a control storage page size of 64 words and the curve of Figure 5.3.3 (Table 5.3.7) is for a control storage page size of 32 words. The curves of Figure 5.3.1, 5.3.2, and 5.3.3 are plotted together in Figure 5.3.4. The reduction in page size results in the inclusion of the no hardware option point in Figure 5.3.2 (Table 5.3.6) and Figure 5.3.3 (Table 5.3.7). The exclusion of the no hardware option case in Figure 5.3.1 (Table 5.3.5) occurred because the number of words of control required by the no hardware option case is just slightly over a page boundary for 128 word pages. The addition of hardware option 2 causes a reduction in the cost of control storage by one page (128 words) which more than offsets the cost of hardware option 2. The reduction in page size can be generally said to reduce the total cost of any set of hardware options.



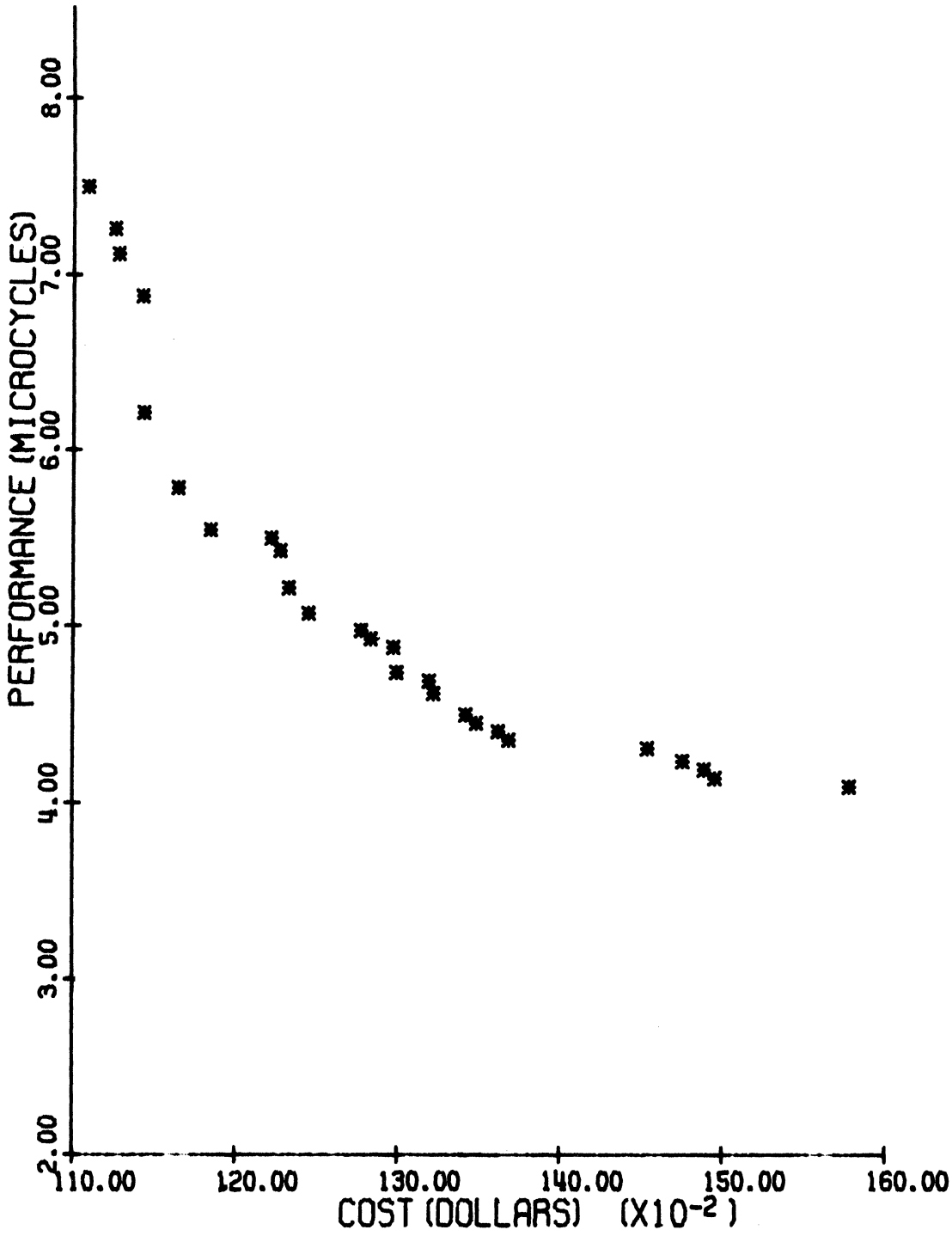

Optimum Cost Performance Curve
Control Storage Page Size 64 Words

Figure 5.3.2

| PERFORMANCE (AVERAGE MICROCYCLES) | COST (DOLLARS) | HARDWARE OPTIONS |
|---|---|---|
| 7.500 | 11152.00 | |
| 7.119 | 11276.39 | 2 |
| 6.881 | 11425.20 | 2,3 |
| 6.214 | 11496.00 | 8 |
| 5.786 | 11644.80 | 2,8 |
| 5.548 | 11921.60 | 2,3,8 |
| 5.429 | 12274.00 | 2,3,4,8 |
| 5.214 | 12328.00 | 2,6,8 |
| 5.167 | 12528.00 | 1,2,6,7,8 |
| 5.071 | 12532.00 | 2,3,8,9 |
| 4.976 | 12874.00 | 2,4,6,7,8 |
| 4.929 | 12932.00 | 1,2,4,6,7,8 |
| 4.833 | 13020.39 | 2,3,4,7,8,9 |
| 4.786 | 13084.80 | 1,2,3,4,7,8,9 |
| 4.738 | 13092.00 | 2,6,8,9 |
| 4.619 | 13220.39 | 2,4,6,8,9 |
| 4.500 | 13420.39 | 2,4,6,7,8,9 |
| 4.452 | 13484.80 | 1,2,4,6,7,8,9 |
| 4.405 | 13620.39 | 2,3,4,6,7,8,9 |
| 4.357 | 13684.80 | 1,2,3,4,6,7,8,9 |
| 4.309 | 14540.80 | 1,2,3,4,5,6,7,8,9 |
| 4.286 | 14825.20 | 2,4,6,7,8,9,10 |
| 4.238 | 14889.60 | 1,2,4,6,7,8,9,10 |
| 4.190 | 15025.20 | 2,3,4,6,7,8,9,10 |
| 4.143 | 15089.60 | 1,2,3,4,6,7,8,9,10 |
| 4.095 | 15945.60 | 1,2,3,4,5,6,7,8,9,10 |

Optimal Cost Performance and Required Hardware Options
Control Storage Page Size 64 Words

Table 5.3.6



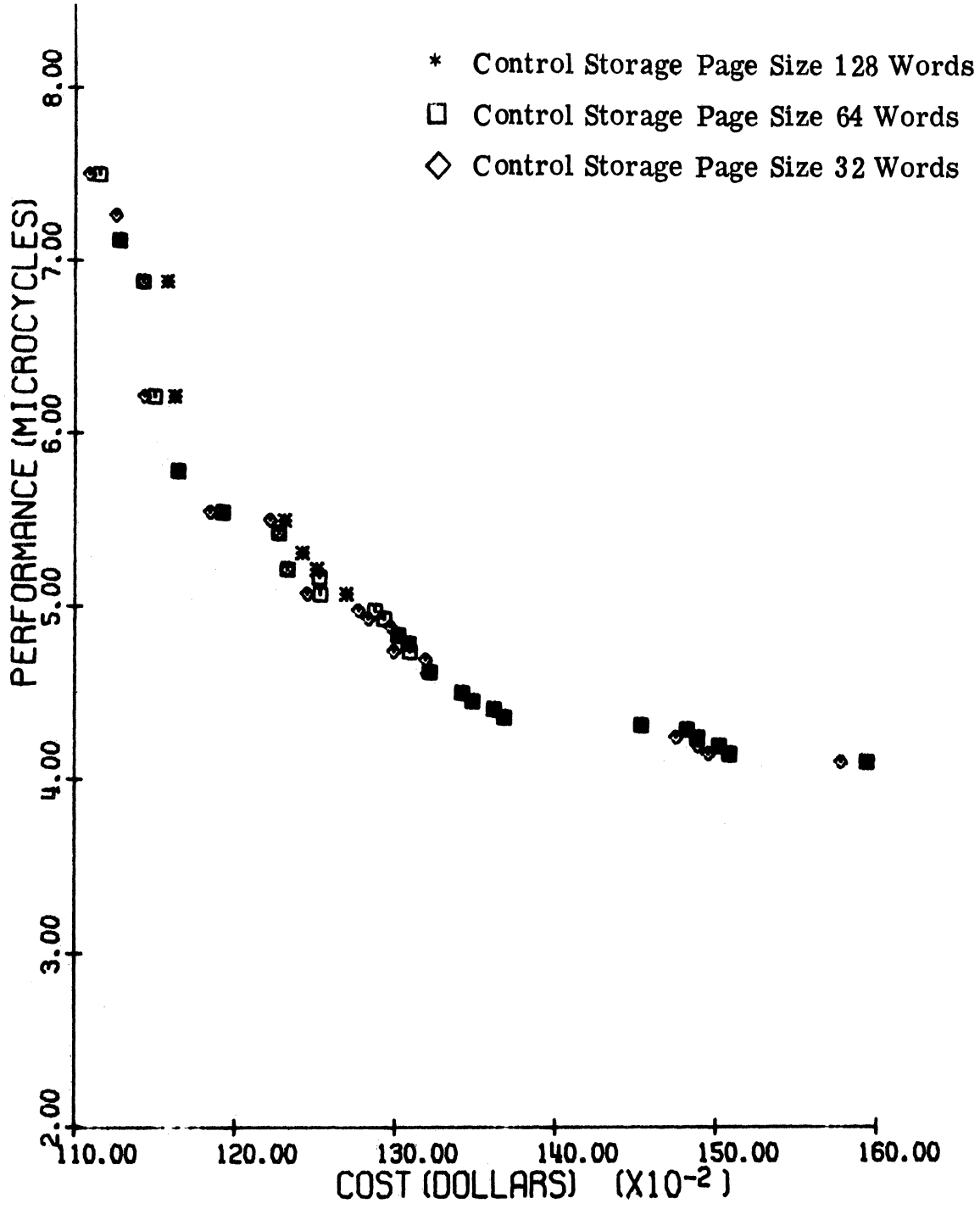

Optimum Cost Performance Curve
Control Storage Page Size 32 Words

Figure 5.3.3

| PERFORMANCE (AVERAGE MICROCYCLES) | COST (DOLLARS) | HARDWARE OPTIONS |
|---|---|---|
| 7.500 | 11088.00 | |
| 7.262 | 11256.00 | 3 |
| 7.119 | 11276.39 | 2 |
| 6.881 | 11425.20 | 2,3 |
| 6.214 | 11432.00 | 8 |
| 5.786 | 11644.80 | 2,8 |
| 5.548 | 11844.80 | 2,3,8 |
| 5.500 | 12220.80 | 1,2,3,7,8 |
| 5.429 | 12274.00 | 2,3,4,8 |
| 5.214 | 12328.00 | 2,6,8 |
| 5.071 | 12448.80 | 2,3,8,9 |
| 4.976 | 12771.60 | 2,4,6,7,8 |
| 4.929 | 12832.80 | 1,2,4,6,7,8 |
| 4.881 | 12971.60 | 2,3,4,6,7,8 |
| 4.738 | 12992.80 | 2,6,8,9 |
| 4.690 | 13192.80 | 1,2,6,7,8,9 |
| 4.619 | 13220.39 | 2,4,6,8,9 |
| 4.500 | 13420.39 | 2,4,6,7,8,9 |
| 4.452 | 13484.80 | 1,2,4,6,7,8,9 |
| 4.405 | 13620.39 | 2,3,4,6,7,8,9 |
| 4.357 | 13684.80 | 1,2,3,4,6,7,8,9 |
| 4.309 | 14540.80 | 1,2,3,4,5,6,7,8,9 |
| 4.238 | 14758.39 | 1,2,4,6,7,8,9,10 |
| 4.190 | 14890.80 | 2,3,4,6,7,8,9,10 |
| 4.143 | 14958.39 | 1,2,3,4,6,7,8,9,10 |
| 4.095 | 15782.39 | 1,2,3,4,5,6,7,8,9,10 |

Optimum Cost Performance and Required Hardware Options
Control Storage Page Size 32 Words

Table 5.3.7



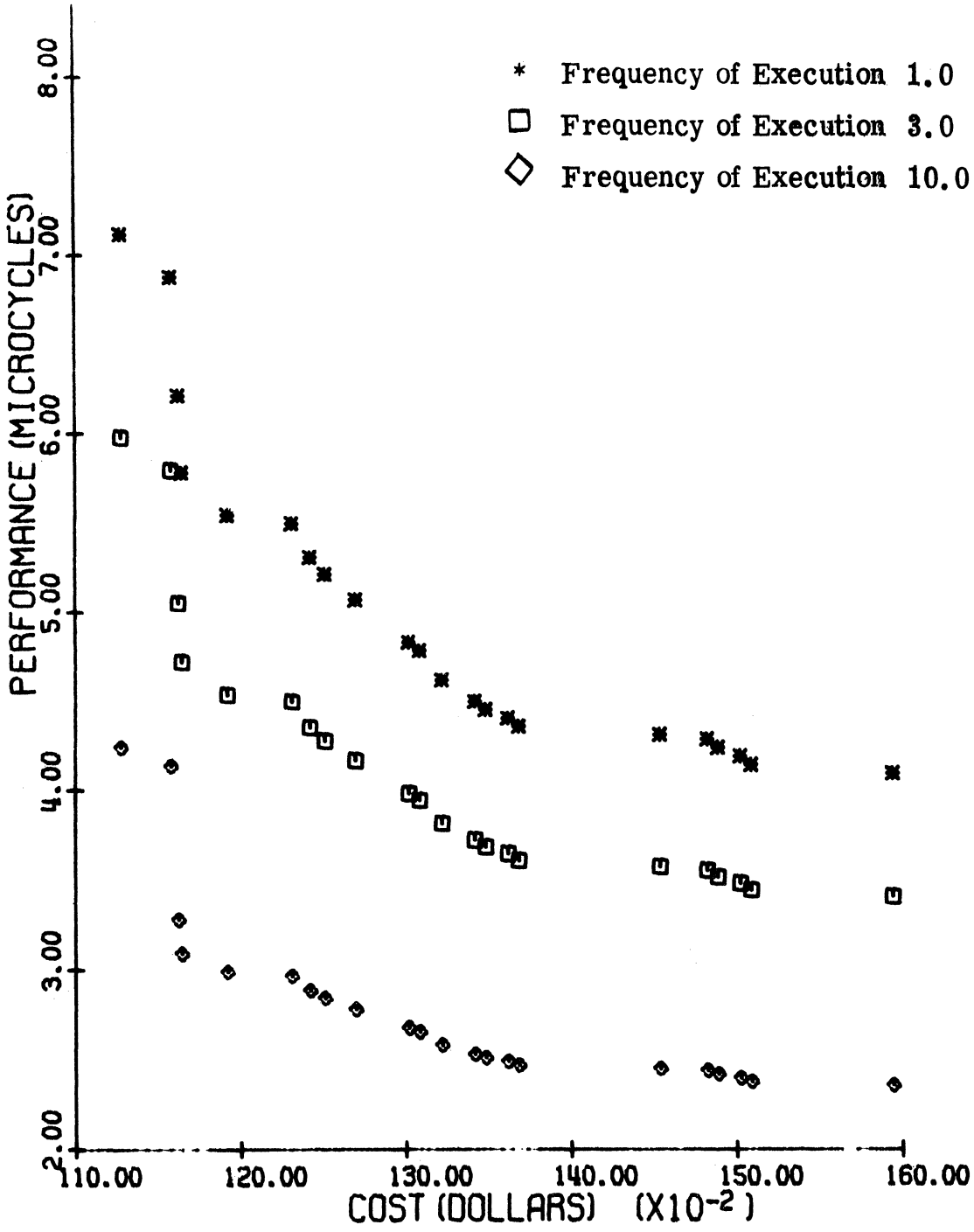

Optimum Cost Performance Curves
Control Storage Page Size 128, 64, 32 Words

Figure 5.3.4

Frequency of Execution

The curves developed in Figure 5.3.5 (5.3.5, 5.3.8, 5.3.9) and Figure 5.3.6 (Tables 5.3.5, 5.3.10, 5.3.11) explore the effect of varying the frequency of execution of different sets of program types. The curves of Figure 5.3.5 are developed by varying the frequency of execution of program types 1, 2, and 3 of Table 5.3.1. It is interesting to note that the curves for an execution frequency of 1.0(*), 3.0(Δ), and 10.0 (□), although different in performance are identical (point by point) in terms of cost and required hardware options. This result occurs because the three program types (1,2,3) all use the same hardware option.

The curves of Figure 5.3.6 are developed by varying the frequency of execution of program types 19, 20, and 21 of Table 5.3.1. The curves of Figure 5.3.6 each contain different points in terms of required hardware options. The curve which has the highest frequency of execution values for program types 19, 20, and 21 of Table 5.3.1 has a significantly less number of option points than its two companion curves. Some of the optimum points of the other two curves have more options which can be applied to improve the performance of program types 19, 20, and 21. Therefore, those points tend to dominate as the frequency of execution of options 19, 20, and 21 is increased.




Optimum Cost Performance Curves
Varying Frequency of Execution of Program Types 1, 2, 3

Figure 5.3.5

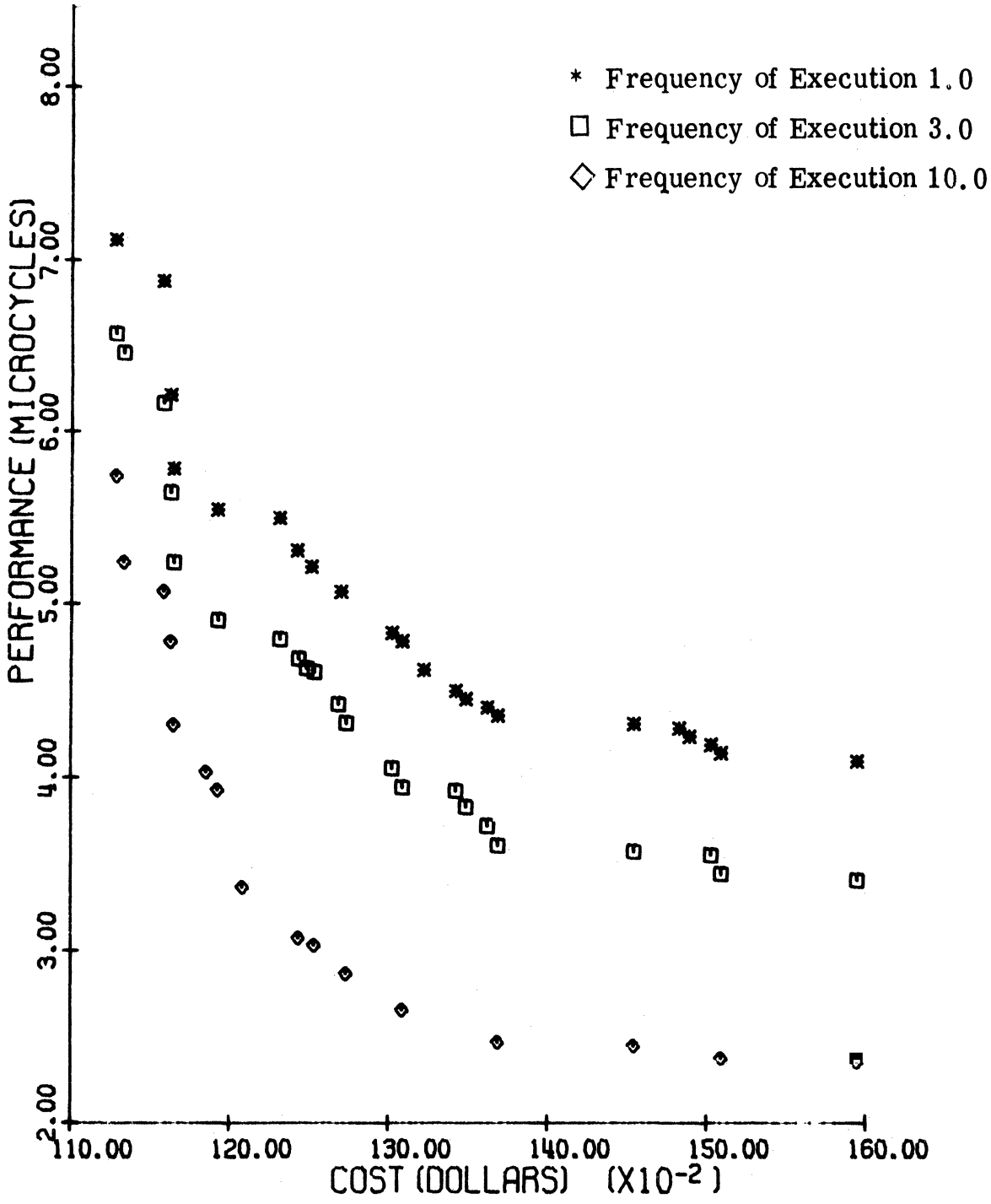
| PERFORMANCE (AVERAGE MICROCYCLES) | COST (DOLLARS) | HARDWARE OPTIONS |
|---|---|---|
| 5.981 | 11276.39 | 2 |
| 5.796 | 11578.80 | 2,3 |
| 5.055 | 11624.00 | 8 |
| 4.722 | 11644.80 | 2,8 |
| 4.537 | 11921.60 | 2,3,8 |
| 4.500 | 12313.60 | 1,2,3,7,8 |
| 4.352 | 12421.60 | 2,8,9 |
| 4.278 | 12513.60 | 2,6,8 |
| 4.167 | 12698.39 | 2,3,8,9 |
| 3.981 | 13020.39 | 2,3,4,7,8,9 |
| 3.944 | 13084.80 | 1,2,3,4,7,8,9 |
| 3.815 | 13220.39 | 2,4,6,8,9 |
| 3.722 | 13420.39 | 2,4,6,7,8,9 |
| 3.685 | 13484.80 | 1,2,4,6,7,8,9 |
| 3.648 | 13620.39 | 2,3,4,6,7,8,9 |
| 3.611 | 13684.80 | 1,2,3,4,6,7,8,9 |
| 3.574 | 14540.80 | 1,2,3,4,5,6,7,8,9 |
| 3.556 | 14825.20 | 2,4,6,7,8,9,10 |
| 3.518 | 14889.60 | 1,2,4,6,7,8,9,10 |
| 3.481 | 15025.20 | 2,3,4,6,7,8,9,10 |
| 3.444 | 15089.60 | 1,2,3,4,6,7,8,9,10 |
| 3.407 | 15945.60 | 1,2,3,4,5,6,7,8,9,10 |

Optimal Cost Performance and Required Hardware Options
Frequency of Execution Program Types 1,2,3 3.0

Table 5.3.8

| PERFORMANCE (AVERAGE MICROCYCLES) | COST (DOLLARS) | HARDWARE OPTIONS |
|---|---|---|
| 4.240 | 11276.39 | 2 |
| 4.135 | 11578.80 | 2,3 |
| 3.281 | 11624.00 | 8 |
| 3.094 | 11644.80 | 2,8 |
| 2.990 | 11921.60 | 2,3,8 |
| 2.969 | 12313.60 | 1,2,3,7,8 |
| 2.885 | 12421.60 | 2,8,9 |
| 2.844 | 12513.60 | 2,6,8 |
| 2.781 | 12698.39 | 2,3,8,9 |
| 2.677 | 13020.39 | 2,3,4,7,8,9 |
| 2.656 | 13084.80 | 1,2,3,4,7,8,9 |
| 2.583 | 13220.39 | 2,4,6,8,9 |
| 2.531 | 13420.39 | 2,4,6,7,8,9 |
| 2.510 | 13484.80 | 1,2,4,6,7,8,9 |
| 2.490 | 13620.39 | 2,3,4,6,7,8,9 |
| 2.469 | 13684.80 | 1,2,3,4,6,7,8,9 |
| 2.448 | 14540.80 | 1,2,3,4,5,6,7,8,9 |
| 2.437 | 14825.20 | 2,4,6,7,8,9,10 |
| 2.417 | 14889.60 | 1,2,4,6,7,8,9,10 |
| 2.396 | 15025.20 | 2,3,4,6,7,8,9,10 |
| 2.375 | 15089.60 | 1,2,3,4,6,7,8,9,10 |
| 2.354 | 15945.60 | 1,2,3,4,5,6,7,8,9,10 |

Optimal Cost Performance and Required Hardware Options
Frequency of Execution Program Types 1,2,3 10.0.
Table 5.3.9




Optimum Cost Performance Curves
Varying Frequency of Execution of Program Types 19, 20, 21

Figure 5.3.6

| PERFORMANCE (AVERAGE MICROCYCLES) | COST (DOLLARS) | HARDWARE OPTIONS |
|---|---|---|
| 6.574 | 11276.39 | 2 |
| 6.463 | 11326.39 | 3 |
| 6.167 | 11578.80 | 2,3 |
| 5.648 | 11624.00 | 8 |
| 5.241 | 11644.80 | 2,8 |
| 4.907 | 11921.60 | 2,3,8 |
| 4.796 | 12313.60 | 1,2,3,7,8 |
| 4.685 | 12430.39 | 1,2,3,4,6,7 |
| 4.630 | 12478.80 | 2,3,4,8 |
| 4.611 | 12530.39 | 1,3,4,7,8 |
| 4.426 | 12678.80 | 2,3,4,7,8 |
| 4.315 | 12730.39 | 1,2,3,4,7,8 |
| 4.056 | 13020.39 | 2,3,4,7,8,9 |
| 3.944 | 13084.80 | 1,2,3,4,7,8,9 |
| 3.926 | 13420.39 | 2,3,4,6,8,9 |
| 3.833 | 13484.80 | 1,2,4,6,7,8,9 |
| 3.722 | 13620.39 | 2,3,4,6,7,8,9 |
| 3.611 | 13684.80 | 1,2,3,4,6,7,8,9 |
| 3.574 | 14540.80 | 1,2,3,4,5,6,7,8,9 |
| 3.556 | 15025.20 | 2,3,4,6,7,8,9,10 |
| 3.444 | 15089.60 | 1,2,3,4,6,7,8,9,10 |
| 3.407 | 15945.60 | 1,2,3,4,5,6,7,8,9,10 |

Optimum Cost Performance and Required Hardware Options
Frequency of Execution Program Types 19, 20, 21 3.0
Table 5.3.10

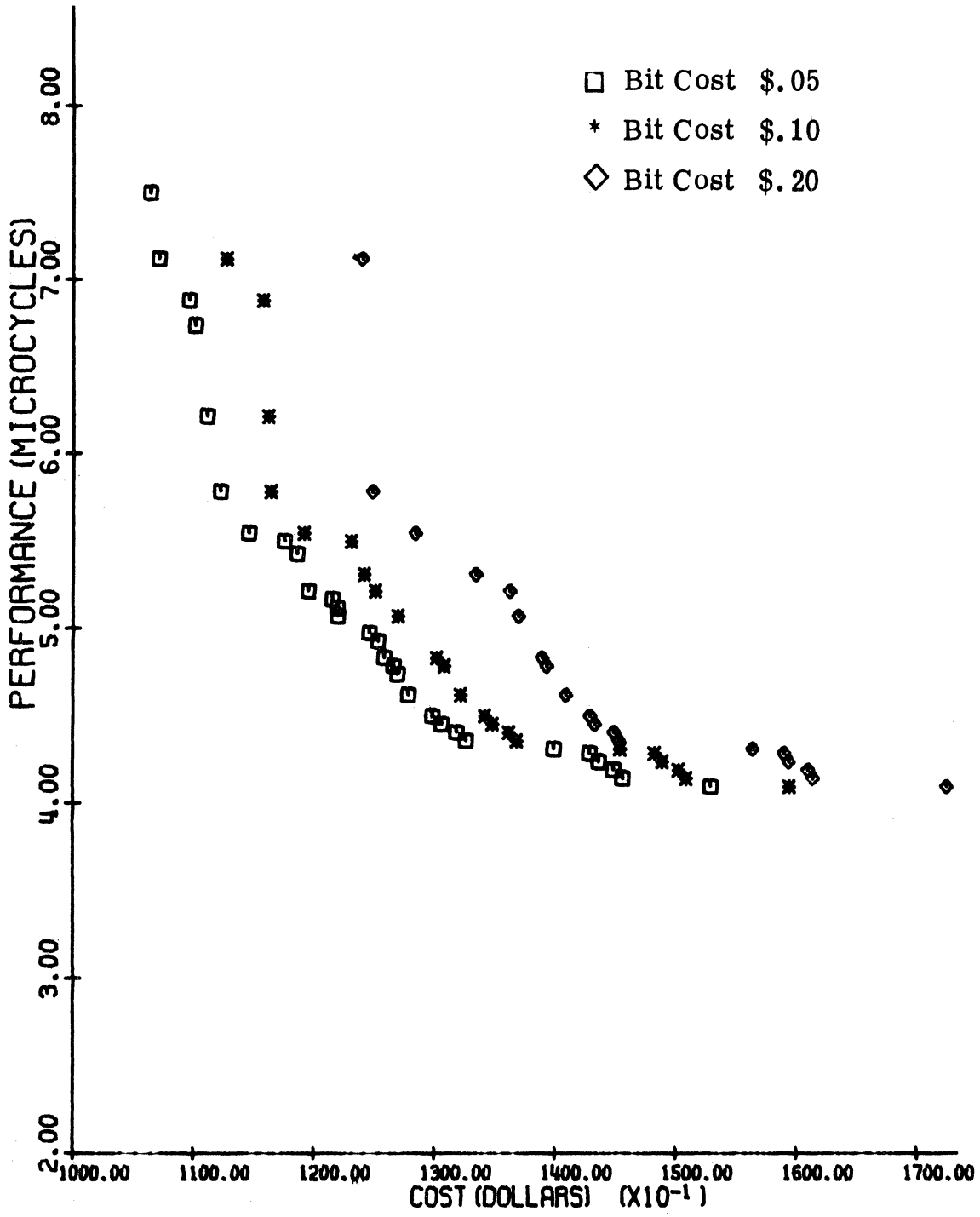
| PERFORMANCE (AVERAGE MICROCYCLES) | COST (DOLLARS) | HARDWARE OPTIONS |
|---|---|---|
| 5.740 | 11276.39 | 2 |
| 5.240 | 11326.39 | 3 |
| 5.073 | 11578.80 | 2,3 |
| 4.781 | 11624.00 | 8 |
| 4.302 | 11644.80 | 3,8 |
| 4.031 | 11850.39 | 1,2,3,4 |
| 3.927 | 11921.60 | 2,3,8 |
| 3.365 | 12080.39 | 1,2,3,4,7 |
| 3.073 | 12430.39 | 1,2,3,4,6,7 |
| 3.031 | 12530.39 | 1,3,4,7,8 |
| 2.865 | 12730.39 | 1,2,3,4,7,8 |
| 2.656 | 13084.80 | 1,2,3,4,7,8,9 |
| 2.469 | 13684.80 | 1,2,3,4,6,7,8,9 |
| 2.448 | 14540.80 | 1,2,3,4,5,6,7,8,9 |
| 2.375 | 15089.60 | 1,2,3,4,6,7,8,9,10 |
| 2.354 | 15945.60 | 1,2,3,4,5,6,7,8,9,10 |

Optimum Cost Performance and Required Hardware Options
Frequency of Execution Program Types 19, 20, 21 10.0

Table 5.3.11

Control Storage Bit Cost

The curves of Figure 5.3.7 are developed by varying the cost of a bit of control storage. The cost is varied from five cents (Table 5.3.12) to 10 cents (Table 5.3.5) to 20 cents (Table 5.3.13) per bit. As would be expected the curves are affected more drastically at the low cost performance end of the curves. As the cost of a bit of control storage increases many of the optimum cost performance points drop out. This effect is noticed because improved performance and a decrease in the length of control storage generally occur together. Thus as the cost of control storage increase marginal cost performance points are dropped.




Optimum Cost Performance Curves
Control Storage Bit Costs $.05, $.10, $.20

Figure 5.3.7

| PERFORMANCE (AVERAGE MICROCYCLES) | COST (DOLLARS) | HARDWARE OPTIONS |
|---|---|---|
| 7.500 | 10640.00 | |
| 7.119 | 10713.20 | 2 |
| 6.881 | 10964.39 | 2,3 |
| 6.738 | 11015.60 | 6 |
| 6.214 | 11112.00 | 8 |
| 5.786 | 11222.39 | 2,8 |
| 5.548 | 11460.80 | 2,3,8 |
| 5.500 | 11756.80 | 1,2,3,7,8 |
| 5.429 | 11864.39 | 2,3,4,8 |
| 5.214 | 11956.80 | 2,6,8 |
| 5.167 | 12156.80 | 1,2,6,7,8 |
| 5.119 | 12195.20 | 2,3,6,8 |
| 5.071 | 12199.20 | 2,3,8,9 |
| 4.976 | 12464.39 | 2,4,6,7,8 |
| 4.929 | 12535.20 | 1,2,4,6,7,8 |
| 4.833 | 12585.20 | 2,3,4,7,8,9 |
| 4.786 | 12662.39 | 1,2,3,4,7,8,9 |
| 4.738 | 12695.20 | 2,6,8,9 |
| 4.619 | 12785.20 | 2,4,6,8,9 |
| 4.500 | 12985.20 | 2,4,6,7,8,9 |
| 4.452 | 13062.39 | 1,2,4,6,7,8,9 |
| 4.405 | 13185.20 | 2,3,4,6,7,8,9 |
| 4.357 | 13262.39 | 1,2,3,4,6,7,8,9 |
| 4.309 | 13990.39 | 1,2,3,4,5,6,7,8,9 |
| 4.286 | 14287.60 | 2,4,6,7,8,9,10 |
| 4.238 | 14364.80 | 1,2,4,6,7,8,9,10 |
| 4.190 | 14487.60 | 2,3,4,6,7,8,9,10 |
| 4.143 | 14564.80 | 1,2,3,4,6,7,8,9,10 |
| 4.095 | 15292.80 | 1,2,3,4,5,6,7,8,9,10 |

Optimum Cost Performance and Required Hardware Options
Control Storage Bit Cost $.05

Table 5.3.12

| PERFORMANCE (AVERAGE MICROCYCLES) | COST (DOLLARS) | HARDWARE OPTIONS |
|---|---|---|
| 7.119 | 12402.80 | 2 |
| 5.786 | 12489.60 | 2,8 |
| 5.548 | 12843.20 | 2,3,8 |
| 5.309 | 13343.20 | 2,8,9 |
| 5.214 | 13627.20 | 2,6,8 |
| 5.071 | 13696.80 | 2,3,8,9 |
| 4.833 | 13890.80 | 2,3,4,7,8,9 |
| 4.786 | 13929.60 | 1,2,3,4,7,8,9 |
| 4.619 | 14090.80 | 2,4,6,8,9 |
| 4.500 | 14290.80 | 2,4,6,7,8,9 |
| 4.452 | 14329.60 | 1,2,4,6,7,8,9 |
| 4.405 | 14490.80 | 2,3,4,6,7,8,9 |
| 4.357 | 14529.60 | 1,2,3,4,6,7,8,9 |
| 4.309 | 15641.60 | 1,2,3,4,5,6,7,8,9 |
| 4.286 | 15900.39 | 2,4,6,7,8,9,10 |
| 4.238 | 15939.20 | 1,2,4,6,7,8,9,10 |
| 4.190 | 16100.39 | 2,3,4,6,7,8,9,10 |
| 4.143 | 16139.20 | 1,2,3,4,6,7,8,9,10 |
| 4.095 | 17251.20 | 1,2,3,4,5,6,7,8,9,10 |

Optimum Cost Performance and Required Hardware Options
Control Storage Bit Cost $.20

Table 5.3.13

# Chapter VI

# CONCLUSION

The contributions of the research are both general and specific. The general contribution is the design method and the specific contributions are the elements which were developed to implement the design method.

The design method which has been developed in this thesis has a number of desirable features. First, the design method is directed toward a CPU design environment which is rapidly coming into existence. Second, the design method is user directed. A very determined effort was made to let the user specify what functions are required for his computer applications. A large set of operations (called program types) were selected and combined into a language (PTL). The user writes his programs or translates his programs into PTL. It is then very easy to gather statistics on user requirements. Third, the design method makes optimum use of the human designer. The connection between the program types and the CPU model is the implementation methods. There are one or more implementation methods for each program type. Each implementation method specifies a CPU configuration. The human designer specifies the implementation methods and is concerned with performance - cost tradeoffs at the

program type level. The computer, guided by the optimization program, performs the performance - cost tradeoffs for the entire set of program types using the individual implementation methods supplied by the designer. The designer specifies several different configurations for each program type. The designer does not have to attempt to evaluate the combined effect of adding an optional piece of hardware on 100 or more program types. He can focus his attention on each program type one by one. The question of automatic or semi-automatic generation of program type implementations which require one or more hardware options using the base hardware implementation is discussed in Appendix D.

The CPU model, PTL, the implementation concept, and the optimization techniques are all specific contributions.

PTL is a machine independent intermediate language. The language is designed to meet two principal objectives. First, the language allows valid statistics to be gathered on the functional requirements of the computer user. Each program type is selected to provide a basic function which is a major building block of a highly used computer area. Many of the program type operations have the potential of direct implementation in hardware with an execution time of one microcycle. Thus, the language should allow valid statistics to be gathered on those basic operations (in terms of basic hardware)

the user wishes to perform. Second, the language is machine independent. The language meets the requirements of an UNCOL language. It is hoped that compilers, interpreters, etc. would be written in PTL and would produce code in PTL. Then a new computer could take advantage of existing compilers, interpreters, user programs, etc. by providing one new compiler which would be a PTL compiler. The need for an UNCOL-like language and the possible advantages of such a language have been discussed for years. However, for one reason or another, an UNCOL language has not been developed. It is time to provide an UNCOL-like language, and PTL is a first attempt at such a language.

The CPU model is very flexible and has a number of good features. The CPU is modeled as a base hardware set plus a set of hardware options. The base hardware set defines a general architecture. Therefore, each design is restricted to a general architecture. However, the hardware options allow major variations of the design about the general architecture. The hardware options consist mainly of function units and the use of the hardware option concept in the model seems to be very realistic in view of the rapid acceptance of large and medium scale integration. The cost functions that the model uses are realistic. The cost of a hardware option is dependent upon the choice of the other hardware options as is the number of

microbits required to control hardware option. The specification of the microinstruction word length is very flexible. The use of both field encoding and word encoding are modeled.

The implementation method concept allows the hardware software tradeoffs to be accomplished at the microcontrol level. The implementation methods could also be used as one input into a microprogram compiler. An additional contribution is the definition of vertical and horizontal microprogramming which are provided in Appendix C. There is currently no standard or well defined definition of either.

The optimization technique involves three distinct parts. First, the use of bounding techniques allows the search procedure to be applied to the hardware option space (H space $< 10^6$) instead of the implementation space (I space $> 10^{12}$). The bounding techniques use the upper bound in performance and a lower bound in the hardware option space. Second, a modification of the Lawler Bell technique is combined with the bounding procedure to eliminate large sections of the hardware option space. Third, a slope selection techique was developed to select the optimal implementation set for any fixed hardware option set. The techique is faster than any with which we are familiar.

There are a number of extensions which could be made to the

work presented here. First, the Program Type Language should be completed by allowing a users group to experiment with and make appropriate changes in the language. Second, a PTL compiler should be developed. The development of a compiler would help to answer many of the questions relevant to the UNCOL concept. Third, the working set idea relative to variables should be explored. Such a concept could be used to determine the optimal amount of local storage. The optimal amount of local storage cannot be determined using the design techniques developed in this dissertation. The optimal amount of storage does not depend upon the number of occurrences of each program type but depends upon the set of active variables and the change in the set of active variables. Thus if the local storage area is full and a new variable is required by the currently executing program type, a memory write and a memory read cycle will be required to acquire the variable. Fourth, a method to optimally select the form of the microinstruction word should be developed. The model which is used in this dissertation considers the cost of the microinstruction word but the form of the word is specified by the designer. It would be useful to develop a method which analyzes the tradeoffs in speed versus control memory size for different microinstruction word forms.

# APPENDIX A

Implementation of High Level Language Instructions in PTL and IBM 360 Model 25 Microcode.

A Fortran DO Loop, a APL submatrix selection, and a SNOBOL pattern match will be implemented in PTL and then converted into IBM 360 model 25 (2025 processor) microcode. The appendix will proceed as follows: 1) The higher level instructions and equivalent PTL code will be given. 2) A list of the PTL program types used in 1 will be compiled. 3) A brief introduction to IBM 360 model 25 (2025) processor) will be given. 4) The IBM 360 model 25 microcode required to implement the program types listed in 2 will be given. 5) The combined microcode required to implement the higher level instructions will be given.

A1 High Level Language Instructions and Equivalent PTL Instructions

FORTRAN DO LOOPS

```
      INTEGER *2    I,J,S1
a)    J = 0
      DO 1 I = 1, S1
1     J = J + I
```

equivalent PTL Code

```
      J ← '0'
      I ← '1'
      DO  S1  L1
      J ← J + I
L1    I ← I + 1
```

```
b)    J = 0
      DO 1 I = 2, S1
1     J = J + I
```

equivalent PTL Code

```
      J ← '0'
      I ← '2'
      S1 ← S1 - '1'
      DO  S1  L1
      J ← J + I
L1    I ← I + 1
```

c)
```
      J = 0
      DO  1  I = 2, S1, 2
1     J = J +1
```

equivalent PTL Code

```
      J ← '0'
      I ← '2'
L1    J ← J + S2
      I ← I + '2'
      BR  S2 ≤ S1   L1
```

---

APL Submatrix Selection

a)
```
      C ← A/B
```

equivalent PTL Code

```
      ADO  L1  ARRAY  B  S1, S2  '12'
      BR  L1  A(S2) = 0
      BR  L2  S2 ≠ 1
      I ← '0'
L2    I ← I + '1'
      C(S1,I) ← B(S1,S2)
L1    CONTINUE
```

b) C← A/[ 1] B

equivalent PTL Code

```
        ADO  L1  ARRAY  B  S1,S2  '2 1'
        BR   L1  A(S1) = '0'
        BR   L2   S1 ≠ 1
         I ← '0'
L2       I ← I + '1'
        C(I,S2) ← B(S1,S2)
L1      CONTINUE
```

---

SNOBOL Pattern Match

a) HERE S1 S2 : F(L1)

equivalent PTL Code

```
HERE  MATCH  S1 in S2 'DUMMY'  L1
```

---

b) HERE S1 S2 : S(L1)

equivalent PTL Code

```
HERE  MATCH  S1 in S2  'DUMMY'  L2
        BR   L1
L2    CONTINUE
```

## A2 List of PTL Operations Used in A1

1) S1 ← '0'

2) S1 ← 'K'

3) S1(S2, S3) ← S4(S5, S6)

4) INCR S1

5) S1 ← S2 + '2'

6) S1 ← S2 + S3

7) DO S1 S2 L1

8) ADO L1 ARRAY S1 S2, S3 'C'

9) BR S1 $\leq$ '10' L1

10) BR S1 = '0' L1

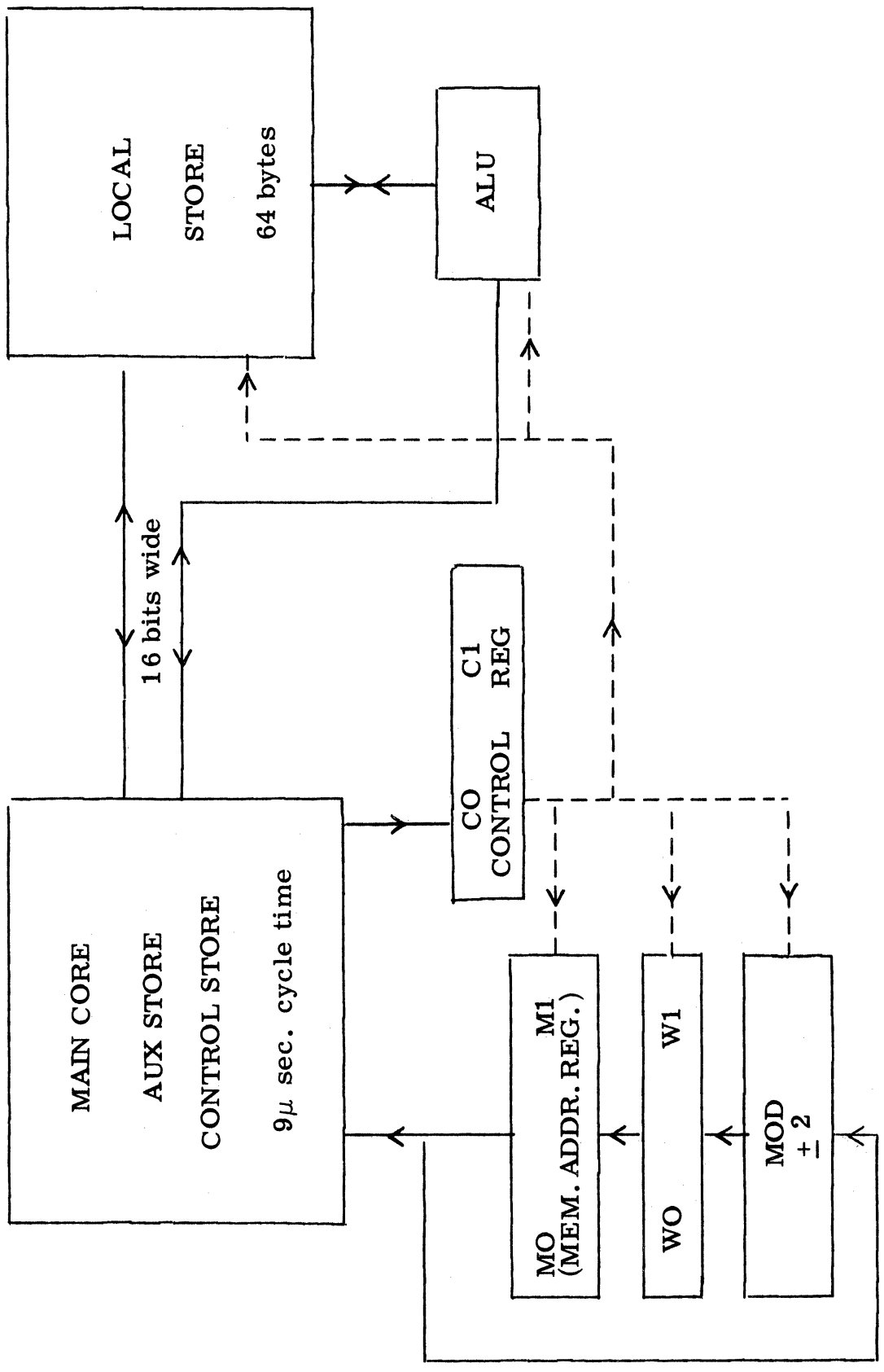
11) BR S1 $\neq$ 'C' L1

12) MATCH S1 IN S2 S3 L1

## A3 IBM 360 2025 Processor (Model 25)

The IBM Model 2025 processor (I1) is a microprogrammable central processing unit having limited capabilities. The ALU hardware is byte oriented and can perform 1 arithmetic or 1 logical operation in 1 main memory cycle time. The main memory cycle time is 0.9 usec and words are accessed in halfword (16 bit) increments. The microcontrol storage, special storage areas, and program storage are all contained in main memory. Since microcontrol words are held in the main memory, any microoperation which requires data from main memory takes two memory cycles.

If the I/O portion of the 2025 is ignored, the CPU and its connection to main storage can be represented as shown in Figure A1. The M register, W register and Mod $\pm 2$ hardware are used to sequence the microprogram. The sequencing of the microprogram can be handled concurrently with ALU operations. The ALU operates on data in the Local Store (LS). The ALU performs logical and arithmetic operations upon bytes not half words. Thus the addition of two half words in LS will require the execution of two microinstructions. It is possible to route a LS (2 bytes) through the mod $\pm 2$ hardware in one microcycle. The instruction types are explained and listed in Figure A2 and several examples of typical instructions are listed in Figure A3.




Model 2025 Processor Simplified Block Diagram
Figure A1

Major Word Type

Type 0 Set/Reset

i) set/reset s0 for ± function

ii) set mode for local store addressing

1 Arithmetic Constant

An arithmetic operation between a byte of LS and a 4 bit constant (within the control word) is performed. The result of written back into the same LS byte or placed on the Z bus. The constant can be used as high, low or both 8 bits and would be represented as K 88, K08, K80, etc. operations include +, AND, XOR, OR, -, COMP AND.

2 Storage Word

i) Read (Write) half word or byte into or out of main core. Main core (not AUX STORE or CONTROL STORE) can be addressed indirectly (LS).

ii) Update indirect address (LS).

iii) Move half words from LS to LS.

3 Move/Arithmetic

An arithmetic operation is performed between two byte of LS and stored back into one of the two, also Z conditions are set.

Word Type
Figure A2

4 Branch Unconditional

A branch to another part of control store is executed with the return address stored in LS zone 4 (I register)

5 Branch on Mask

i) Does 4, 8, 16 way branch upon test of bits (mask) either dynamic test or LS byte.

ii) 2 way branch upon test of LS byte for all o's

6/7 Branch on Condition Word

Tests a single bit of a LS byte for 1 or 0 branch if condition next, next sequential word, if condition not meet.

Word Type

Figure A2 (Continued)

1) U1 = U1 + H0

Bytes U1 and H0 are added and stored in U1

2) U1C = U1 + H0

Bytes U1 and H0 are added and stored in U1 the carry flip flop is set if a carry occurs.

3) U1 = U1 + H0 + C

Bytes U1 and H0 and the carry flip flop input are added and the result stored in U1

4) U = U + 2

The half word U is incremented by 2.

5) V1 = V1 + KAA

The constant AA is added to V1 and stored in V1

6) V1 = V1 $ K04

The constant 4 is ORed to V1 and stored in V1. Note that constants may take the form KAA, KOA, KAO, but not the form KAB.

7) RDH U V

The contents of main memory (half word) addressed by V is placed in U.

8) RDH U V + 2

Same as 7 but V is incremented by 2 after the read.

LS Storage Locations are designated as U, V, G, D, I, T, P, H; the right half of a word is designated as U0, the left half of U1

Typical IBM 360 2025 Processor Microcode
Figure A3

9) LABEL BR IF H1 = NZ

A branch to LABEL is executed if H1 is not zero.

10) LABEL BR IF G1 BIT 3 = 0

A branch to LABEL is executed if Bit 3 of G1 is zero.

11) LABEL BR IF C = 0

A branch to LABEL is executed if the carry flip flop is zero.

Typical IBM 360 2025 Processor Microcode
Figure A3 (Continued)

## A4 IBM 360 2025 Processor Microcode to Implement Instructions Listed in A2

1) S1 ← '0' — S1 assigned to D

```
D0 = 0 $ K00
D1 = 0 $ K00
```

2) S1 ← 'C' — S1 assigned to D

```
D0 = 0 $ KLC1
D0 = 0 $ KHC2
D1 = 0 $ KLC3
D1 = 0 $ KHC4
```

3) S1 (S2, S3) ← S4 (S5, S6)

Assume that each array is stored by rows. The location of S1 is held in LS location U. The first location of S1 gives the number of rows, the second location gives the number of elements in each row, the third location points to the start of the first row, the fourth location points to the start of the second row, etc. S4 is held in V, S2 in G, S3 in D, S5 in I, and S6 in T.

```
POC = I0 + I0
P1 = I1 + I1 + C          P ← 2 * S5
POC = P0 + V0
```

| | |
|---|---|
| P1 = P1 + V1 + C | P = S4 + 2 * S5 |
| P = P + 2 | |
| P = P + 2 | |
| RDH P P | GET LOC OF S4(S5,*) |
| P0C = P0 + T0 | |
| P1C = P1 + T1 + C | |
| P0C = P0 + T0 | |
| P1C = P1 + T1 + C | GET LOC OF S4 (S5,S6) |
| RDH P P | GET S4(S5,S6) |
| H0C = G0 + G0 | |
| H1 = G1 + G1 + C | H ← 2 * S2 |
| H0C = G0 + U0 | |
| H1C = G1 + U1 + C | H = S1 + 2 * S2 |
| H = H + 2 | |
| RDH H H | GET LOC OF S1(S2,*) |
| H0C = D0 + H0 | |
| H1 = D1 + H1 + C | |
| H0C = D0 + H0 | |
| H1 = D1 + H1 + C | GET LOC OF S1(S2,S3) |
| STH H P | STOR S4(S5,S6)→S1(S2,S3) |

4) INCL S1 — S1 assigned to U

```
U = U + 1
```

5) S1 ← S2 + 2 — S1 assigned to U, S2 to D

```
U = D

U = D + 2
```

6) S1 ← S2 + S3 — S1 assigned to U, S2 to D; S3 assigned to G

```
U = D

U0C = U0 + G0

U1 = U1 + G1 + C
```

7) D0 S1 L1 — S1 assigned to H

```
L1   LOOP  CODE
       :     :
     LOOP CODE

     H = H -1

     L1   BR   H   NZ
```

8) ADO L1 ARRAY S1 S2,S3 '12'

Assign S1, S2, S3 to G, U, V, respectively. Assume array, are stored as specified in Example 3.

```
     U0 = K01

     U1 = K00
L2   V0 = K01

     V1 = K00
```

```
START  LOOP  CODE
         :
       LOOP  CODE
       V = V + 1
       G = G + 2
       RDH  H  G-2
       H0 = H0 ⊕ V0                TEST  IF
       START  BR  H0  H2           END  OF
       H1 = H1 ⊕ V1                ROW  HAS  BEEN
       START  BR  H1  NZ           REACHED
       RDH  H  G
       U = U + 1
       H0 = H0 ⊕ U0                TEST  IF
       L2  BR  H0  NZ              LAST  ROW
       H1 = H1 ⊕ U1                HAS  BEEN
       L2  BR  H1  NZ              PROCESSED
  L1   CONTINUE
9)     BR  S1 ≤ S2  L1             assign S2 to, U, S1 to D
         U0C = U0 - D0
         U1 = U1 - D1 - C
         L1  BR  U18  Z            L1  IF NO OVERFLOW
```

10) BR S1 = '0' L1 assigns S1 to H

L1 BR H NZ

11) BR S1 ≠ S2 L1 assign S1 to U, S2 to D

H0 = U0 ⊕ D0

L1 BR H0 NZ

H0 = U1 ⊕ D1

L1 BR H0 NZ

12) MATCH S1 IN S2 L2

Assume that U points to S1, V to S2, that the first location S1 and S2 contain the length of each. In this example string lengths are restricted to 255 words or less. and the location of the match is not required.

```
    RDB   G0   U + 1          LENGTH S1
    BDB   G1   V + 1          LENGTH S2
L1  RDB   I0   U              FIRST CHARACTER S1
L3  RDB   I1   V + 1
    H0  = I0 ⊕ I2
    L4 BR H0 Z
    G1 = G1 - 1
    L2 BR P0 Z                NO MATCH
    BR L3
```

```
L4      DO = G0 - 1
        L6  BR  D0  Z                   MATCH
        D1 = U + 1
        P0 = G1 - 1
        L2  BR  P0  Z
        T1 = V
L5      RDB  I0  U + 1
        RDB  I1  V + 1
        P0 = I0 ⊕ I1
        L1  BR  P0  NZ
        D0 = D0 - 1
        L6  BR  D0  Z                   MATCH
        P0 = P0 - 1
        L2  BR  P0  NZ                  NO MATCH
L6      CONTINUE
```

A5 Microcode Required to Implement Higher Level Instructions of Section A1.

The instructions of section A1 will be implemented in IBM 2025 processor microcode. The higher level instructions were translated into PTL code. Each line of PTL code will be replaced by the equivalent microcode which was developed in Section A4. The variable assignment will have to be modified in some cases.

## FORTRAN DO LOOPS

a)

```
      J = 0
      DO  1  I = 1, S1
    1 J = J + 1
```

equivalent PTL Code

```
      I ← '1'
      J ← '0'
      DO  S1  L1
      J ← J + I
L1    I ← I + 1
```

equivalent microcode

Assign I to U, J to V, S1 to H

```
      U0 = K01                    I
      U1 = K00                    I = 1
      V0 = K00                    J
      V1 = K00                    J = 1
L1    V0C = V0 + U0
      V1 = V1 + U1 + C            J = J + 1
      U = U + 1                   I = I + 1
      H = H - 1                   S1 = S1 - 1
      L1  BR  H  NZ               LOOP TEST
```

b)
```
        J = 0
        DO  1   I = 2, S1
1       J = J + I
```

equivalent PTL Code

```
        J = '0'
        I = '2'
        S1 = S1 - '1'
        DO   S1   L1
        J ← J  + I
L1      I ← I + 1
```

equivalent microcode

```
        assign J to U, I to V, S1 to H
        U0 = K00
        U1 = K00                        J = 0
        V0 = K02
        V1 = K00                        I = 2
        H = H - 1                       S1 = S1 - 1
L1      U0C = U0 + V0
        U1 = U1 + V1 + C                J = J + J
        V = V + 1                       I = J + 1
        H = H - 1
        L1   BR   H   NZ                LOOP TEST
```

c)
```
      J = 0
      DO 1  I = 2, S1, 2
1     J = J + I
```

equivalent PTL Code

```
      J ← '0'
      I ← '2'
L1    J ← J + I
      I ← I + '2'
      BR  I ≤ S1  L1
```

equivalent microcode

```
      assign J to U, I to V, S1 to G
      U0 = K00
      U1 = K00                    J ← 0
      V0 = K02
      V1 = K00                    I ← 2
L1    U0C = U0 + V0
      U1 = U1 + V1 + C            J ← J + I
      V = V + 2                   I = I + 2
      H = G
      H0C = H0 - V0
      H1C = H1 - V1 + C           S1 - I
      L1  BR  H1  8  Z            BR  L1 +
```

APL Submatrix Selection

c) C ← A/B

equivalent PTL Code

```
        ADO  L1  ARRAY  B  S1, S2   '12'
        BR   L1  A(S2) = 0
        BR   L2  S2 ≠ 1
        I ← '0'
L2      I ← I + '1'
        C(S1, I) ← B(S1, S2)
L1      CONTINUE
```

equivalent microcode

```
        assign B to G, S1 to U, S2 to V, A to T, I to P, C to D
        U0 = K01
        U1 = K00                          S1 = 1
L2      V0 = K01
        V1 = K01                          S2 = 1
START   H0C  = T0 + V0
        H1 = T1 + V1 + C
        RDH  H  H                         GET  A(2)
        L5  BR  H0  NZ
        L1  BR  H1  Z                     BR  L1  A(S2) = 0
```

```
L5      H = V
        H0 = H0 ⊕ K01
        L2  BR  H0  NZ
        H1 = K1 ⊕ K00
        L2  BR  H0  NZ                  BR  L2  S2 ≠ 1
        P0 = K00
        P1 = P0                         I ← 0
L2      P = P + 1                       I ← I + 1
        H = U
        H0C = H0 + H0
        H1 = H1 + H1 + C                H ← 2 * S1
        H0C = H0 + G0
        H1 = H1 + G1 + C                H = B + 2 * S1
        H = H + 2
        H = H + 2
        RDH  H  H                       GET LOC B(S1,*)
        H0C = H0 + V0
        H1 = H1 + V1 + C
        H0C = H0 + V0
        H1 = H1 + V1 + C                GET LOC B(S1,S2)
        RDH  H  H                       GET B(S1,S2)
```

```
I = U
I0C = I0 + I0
I1 = I1 + I1 + C                    I = 2 * S1
I0C = I0 + D0
I1 = I1 +D1 + C                     I = C + 2 * S1
I = I + 2
I = I + 2
RDH  I  I                           GET  LOC  C(S1,*)
I0C = I0 + P0
I1 = I1 + P1 + C
I 0C = I0 + P0
I1 = I1 + P1                        GET LOC C(S1,I)
STH  H  I                           C(S1,I) = B(S1,S2)
V = V + 1                           TEST
G = G + 2
RDH  H  G - 2                       IF
H0 = H0 ⊕ V0                        END
START BR  H0  NZ                    OF
H1 = H1 ⊕ V1                        ROW HAS
START  BR  H1  NZ                   BEEN REACHED
RDH  H  G
```

```
        U = U + 1                       TEST IF
        H0 = H0 ⊕ U0                    LAST
        L2  BR  H0  NZ                  ROW HAS
        H1 = H1 ⊕ U1                    BEEN
        L2  BR  H1  NZ                  PROCESSED
L1      CONTINUE
```

b) C ← A/[ 1] B

The microcode is identical to that of a, except that S1 and S2 are interchanged.

SNOBOL Pattern Match

a) HERE S1 S2 : F(L2)

equivalent PTL Code

```
HERE  MATCH  S1 IN S2   'DUMMY'  L2
```

equivalent microcode

Assume that U points to S1, V to S2, that the first location of S1 and S2 contain the length of each. In this example string lengths are restricted to 255 words or less. In this example the location of the match is not required.

```
L4      D0 = G0 - 1
        L6  BR  D0  Z                   MATCH
        D1 = U + 1
        P0 = G1 - 1
        L2  BR  P0  Z                   NO MATCH
        T1 = V
L5      RDB  I0  U + 1
        RDB  I1  V + 1
        P0 = I0 ⊕ I1
        L1  BR  P0  NZ
        D0 = D0 - 1
        L6  BR  D0  Z                   MATCH
        P0 = P0 - 1
        L2  BR  P0  NZ                  NO MATCH
L3      L1  BR                          NO MATCH
L2      CONTINUE
```

# APPENDIX B

The implementations listed below are for the base hardware set of Figure 5. 2. 1. The formats for the microinstruction words are given in Figure 5. 2. 2. Examples of some possible micro-instructions are given in Table 5. 2. 3, and special considerations in using the base microinstructions are listed in Table 5. 2. 4.

1) S3 ← S1 $\wedge$ S2

RB3(S3) ← RB1(S1) $\wedge$ RB2(S2)

LB3(S3) ← LB1(S1) $\wedge$ LB2(S2)

2) S3 ← S1 $\vee$ S2

RB3(S3) ← RB1(S1) $\vee$ RB2(S2)

LB3(S3) ← LB1(S1) $\vee$ LB2(S2)

3) S3 ← ~ S1

RB3(S3) ← $\overline{\text{RB1}}$ (S1)

LB3(S3) ← $\overline{\text{LB1}}$ (S1)

4) S1 ← S2 TR ADDR S3 LABEL

← RB2(LENGTH) + 1 + $\overline{\text{RB1}}$(S2) IN BOUNDS TEST

← LB2(LENGTH) + C + $\overline{\text{LB2}}$(S2)

BRA1 8 LABEL

RB3(T) ← RB1(S3) + RB2(S2)

LB3(T) ← LB1(S3) + C + LB2(S2)

MA ← B(T) MEMC

NOP

NOP

B3(S1) ← MB

BR DONE(T + 1)

5) S1 ← SHIFT S2 CARRY OUT LEFT S3

```
        RB1(S1) ← RB1(S2)
        LB1(S1) ← LB1(S2)

                ← RB1(S3)
        BRAO  4   P1
        LB3(S1) ← RB1(S1)
        RB3(S1) ← '0'
P1              ← RB1(S3)
        BRAO  3   P2
        RB3(S1) ← L1 RB1(S1)
        LB3(S1) ← L1C   LB1(S1)
        RB3(S1) ← L1 RB1(S1)
        LB3(S1) ← L1C   LB1(S1)
        RB3(S1) ← L1 RB1(S1)
        LB3(S1) ← L1C   LB1(S1)
        RB3(S1) ← L1 RB1(S1)
        LB3 (S1) ← L1C  LB1(S1)
P2              ← RB1(S3)
        BRAO  2   P3
        RB3(S1) ← L1 RB1(S1)
        LB3(S1) ← L1C   LB1(S1)
        RB3(S1) ← L1 RB1(S1)
        LB3(S1) ← L1C   LB1(S1)
P3              ← RB1(S3)
        BRAO  2   P4
        RB3(S1) ← L1 RB3(S1)
        LB3(S1) ← L1C   LB3(S1)
P4  BR DONE
```

6) S3 ← S1 + S2

```
RB3(S3) ← RB1(S1) + RB2(S2)
LB3(S3) ← LB1(S1) + C + LB2(S2)
BROF  P1
BR  DONE
P1  SET  OUF
```

7) S3 ← S2 - S1

```
RB3(S3) ← RB2(S2) + 1 + RB1‾(S1)
RB3(S3) ← RB2(S2) + C + RB1‾(S1)
BROF  P1
BR DONE
P1  SET  OUF
```

8) S1 INTEGER TO FLOAT S2

```
RB3(S2L) ← '00₁₆'
         ← LBR(S1)
BRAO   8   L1
LB3(S2L) ← 'C6₁₆'
RB3(S2R) ← RB1‾(S1) + 1
LB3(S2R) ← LB1‾(S1) + C
BR  DONE
L1  LB3(S2L) ← '46₁₆'
RB3(S2R) ← RB1(S1)
LB3(S2R) ← LB1(S1)
```

Stack Operations

There are two locations in S1 which hold pointing information.

i) LENST the current length of the stack

ii) POINST the location (in main memory) of the top of the stake.

9) S1 ← ST LABEL

```
          MA ← B(POINST)        MEMC
RB3(LENST) ← RB3(LENST) + 1 + '1'
LB3(LENST) ← LB3(LENST) + C + '0'   LENST←LENST-1
BRA1   8      L1
RB3(POINST)← RB3(POINST) + 1 + '1'
LB3(POINST)← LB3(POINST) + C + '0'  POINST←POINST-1
B3 (S1)      ← MB
BR  DONE
L1  BR(LENST) ← '0'
    BR  LABEL
```

10) ST ← S1 LABEL

RB1(POINST) ← RB1(POINST) + 1

LB1(POINST) ← LB1(POINST) + C

MA ← B(POINST)

MB ← B(S1) MEMW

RB3(LENST) ← RB3(LENST) + 1

LB3(LENST) ← LB3(LENST) + C

BRAO 2 DONE

RB3(POINST) ← RB2(POINST) + 1 + '$\overline{1}$'

LB3(POINST) ← LB2(POINST) + C + '$\overline{0}$'

RB3(LENST) ← RB2(LENST) + 1 + '$\overline{1}$'

LB3(LENST) ← LB2(LENST) + C + '$\overline{0}$'

BR LABEL

11) CLEAR STACK

RB3(POINST) ← RB2(POINST) + 1 + $\overline{\text{RB1}}$ (LENST)

LB3(POINST) ← LB2(POINST) + C + $\overline{\text{RB1}}$(LENST)

B3 (LENST) ← '0'

Linked List Operations

There are two locations in S1 which hold pointing information.

i) HEAD points to the head of the list

ii) POINTS points to an item of the list, forward pointer.

iii) backward point is at POINTS + 1

12) INCREMENT POINTER LINK LIST S1 LABEL

```
        MA ← B(POINTS)  MEMC
NOP
NOP
BRMB1       16      LABEL          OUTSIDE LIST
B3(POINTS) ← MB
NOP
```

13) DECREMENT POINTER LINK LIST S1 LABEL

```
      RB3(T) ← B1(POINTS) + 1
      LB3(T) ← B1(POINTS) + C
        MA ← B(T)       MEMC
NOP
NOP
BRMB1       16      LABEL          OUTSIDE LIST
B3(POINTS)  ← MB
RB3(POINTS) ← RB2(POINTS) + 1 + '1'
LB3(POINTS) ← LB2(POINTS) + C + '0'
```

14) SET LINK LIST S1

```
        MBL ← 'FF16'
        MA ← B(HEAD)                MEMW
RB3(POINTS) ← RB1(HEAD) + 1
LB3(POINTS) ← LB1(HEAD) + C
        MA ← B(POINTS)              MEMW
RB3(POINTS) ← RB3(HEAD)
LB3(POINTS) ← LB(HEAD)
```

15) EMPTY LINK LIST S1

```
        MA ← B(HEAD)                MEMC
RB3(POINTS) ← RB1(HEAD)
LB3(POINTS) ← LB1(HEAD)
L1  BRMB1   16   DONE
        B(T) ← MB
        MBL ← 'FF16'                MEMW
NOP
NOP
        MA ← B(T)                   MEMR
NOP
BR  L1
```

16) REMOVE S1 LINK LIST LABEL

```
          MA ← B(POINTS)   MEMR   GET FORWARD POINTER
        B(S1) ← MA
    RB3(T1) ← RB1(POINTS) + 1
    LB3(T1) ← LB1(POINTS)+C
        B(T2) ← MB
          MB ← 'FF16'      MEMW   RESET LINK ELEMENT
             ← LB1(T2)
      BRA1 8   L1                 NO FORWARD POINTER
          MA ← B(T1)       MEMR   GET BACKWARD POINTER
RB3(POINTS) ← RB1(T2)+1
LB3(POINTS) ← LB1(T2)+C
   BRMB1 16   LABEL               ERROR REMOVING HEAD
          MA ← B(POINTS)   MEMW
L2  B3(POINTS) ← MB
RB3(POINTS) ← RB2(POINTS)+1+'1'
LB3(POINTS) ← LB2(POINTS)+C+'0'
          MA ← B(POINTS)
          MB ← B(T2)       MEMW
NOP
BR  DONE
L1        MA ← B(T1)       MEMR   GET BACKWARD POINTER
NOP
BR   L2
```

```
17)          INCR    S1

             RB3(S1) ← RB1(S1) + 1
             LB3(S1) ← LB1(S1) + C
             BROF    P1
             BR      DONE
      P1     SET     OVF

18)          DECR    S1

             RB3(S1) ← RB2(S1)+1+'1̄'
             LB3(S1) ← LB2(S1)+C+'0̄'
             BROF    P1
             BR      DONE
      P1     SET     OVF

19)          DO    LABEL    S1

      L1     RB3(S1) ← RB2(S1)+1+'1̄'
             LB3(S1) ← LB2(S1)+C+'0̄'
             BRA1    8    LABEL



             BR      L1
    LABEL    CONTINUE
```

20) BR S1 = S2 LABEL

RB3(T) ← RB2(S1) · EV · RB1(S2)
← RB2(T) + '$\overline{0}$'
BRC L1
LB3(T) ← LB2(S1) · EV · RB1(S2)
← RB2(T) + '$\overline{0}$'
BRC LABEL
L1 BR DONE

21) BR S1 < S2 LABEL

← LB1(S2)
BRA1 8 L1 S2 < 0
← LB1(S1)
BRA1 8 LABEL S1 < 0 S2 > 0
L2 ← RB2(S2) + RB1(S$\overline{1}$)
← LB2(S2) + C+ LB1(S$\overline{1}$) S2 -(S1+1)
BRAO 8 LABEL S2 - S$\overline{1}$
BR DONE
L1 ← LB1(S1)
BRAO 8 DONE
BR L2

DONE

# APPENDIX C

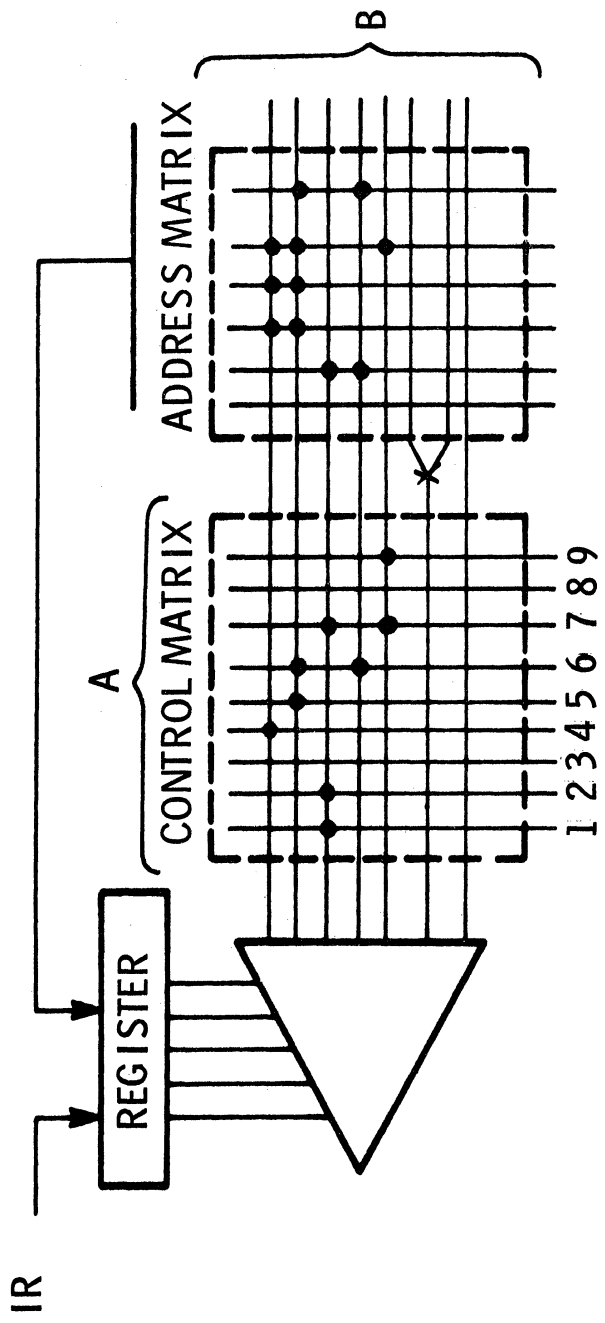
## What is Microprogramming?

M. V. Wilkes [W2,W3] in the early 1950's suggested microprogramming as a more orderly approach to the design of computer control circuitry. He was particularly concerned with the signals needed to control the flow of information among the registers and transformation units that comprise the central processing unit. Wilkes felt that the execution of a machine instruction† by the central processing unit (CPU) could be partitioned into a series of register to register transfers in the CPU. Some of the transfers of data in the CPU would, of course, be made to pass through a transformation unit while going from the source register to the destination register. A typical transformation unit might be capable of shifting, adding, logical disjunction, etc.

A representation of Wilkes' scheme is shown in Figure C.1. with the nonvolatile "control memory" divided into a part A and a part B. Each column of part A represents a microoperation. The excitation of a microoperation causes a predetermined action by one or more elements of the CPU. Some of the actions that a microoperation might cause are: data gated out of a register, data gated into a register, data gated through a transformation unit. Each row in

---

†Machine instruction: the binary code for an operation that is performed by the central processing unit. The code is normally held in the main memory until fetched into the CPU for instruction decoding and execution.




Wilkes' Microprogram Control

Figure C.1

the control memory represents a microinstruction. Part A of the microinstruction controls gating in the central processing unit (CPU) and part B of the microinstruction contains the address of the next microinstruction to be executed. Thus the microprogram controls the gating within the CPU and hence, controls the transfer of data within the CPU.

The control of a central processing unit, according to Wilkes is implemented by the execution of a series of microinstructions. The excitation of the microoperations specified by each microinstruction causes register to register transfers to occur. The series of register to register transfers specified by the microinstructions implement the desired machine instruction.

The question of what is a reasonable choice of microoperations was not clearly specified by Wilkes. The most functionally complete set of microoperations is a set for which each microoperation controls a single gate (input gate or output gate) of one bit of a register. The set is functionally complete in the sense that control is exercised over the smallest unit of storage in the CPU, individual flip-flops. For even moderate sized machines the length of the microinstruction word sufficient to contain all of the microoperations of the preceding set is excessively large. Therefore, it is usually necessary to trade some amount of flexibility (i.e., functional completeness) for a reduction in the microinstruction word size. We will see that

both of the two major implementations of microprogram control result in reduced microinstruction word size.

The terms horizontal microprogramming and vertical microprogramming are used by microprogramming practitioners to form a dichotomy of microinstruction word types. Since there are many different types of microinstructions the definitions that have been developed and which attempt to span the range of microinstruction types are overly generalized. The definitions which are presented below define the end points of the range of microprogram instruction types. The definitions could be said to describe pure horizontal microinstructions and pure vertical microinstructions respectively. In this way the essential difference in philospohy between the two types of microprogramming will be illuminated without the clutter imposed by the constraints of particular applications. After presenting the definitions some of the modifications which are imposed by considerations of cost will be discussed and illustrated. In the remainder of the paper the term horizontal microprogramming will be used to mean more nearly like pure horizontal than pure vertical microprogramming and vice versa for the term vertical microprogramming.

Before defining horizontal and vertical microprogramming the definition of a control point and a control point set will be given.

Def. C.1 Control Point: Each control point for a CPU is a set of control gates with the property that the simultaneous excitation of the gates will cause one of the following actions.

a) The outgatting of data from a register (or portion of a register) onto a data path.

b) The ingatting of data into a register (or portion of a register) from a data path.

c) The routing of data at a bus junction.

d) The selection of the function to be performed by a transformation unit.

Def. C.2 Control Point Set: A control point set for a CPU is a set of control points which partition all the control gates of the CPU.

Each control point is specified by the machine designers and represents a microoperation. The simultaneous excitation of several control points is required to execute a register to register transfer. Horizontal and vertical microprogramming are different methods of encoding the information which specifies the control points

to excite during one microcycle.

Def. C.3 Horizontal microinstruction: A horizontal microinstruction word assigns one bit to each control point of the control set, with the state of the bit controlling the excitation for the corresponding control point.

Def. C.4 Vertical microinstruction: A vertical microinstruction word assigns one bit to several control points of the control set, with the state of the bit controlling the excitation for the corresponding control points.

Horizontal microprogramming condenses word length by allowing each microoperation to control several gates (control point) of a register. Both the width of data paths and the register bit position connection point of data paths are fixed. Therefore the ability to modify data widths is lost, but the ability to select register to register and register to transformation unit data paths remain. Thus a register to register transfer requires the execution of several horizontal microoperations.

A vertical microoperation condenses word length by allowing each microoperation to control several control points. A vertical microoperation may control an entire register to register transfer or the major portion of a register to register transfer. There are, of course, a great number of possible register to register transfers.

The decrease in microinstruction word length is accomplished by implementing a small subset of all the possible register to register transfers and therefore limiting the number of vertical microoperations. Therefore both the ability to modify data path widths and the ability to modify data path connections other than the selection of the preselected data paths are lost.

As was mentioned above, most microprogram implementations are a modification of either horizontal or vertical microprogramming. The most common modification is the use of field encoding. If it is known that certain sets of microoperations are never executed at the same time, then the microorders may be encoded into a field instead of using one bit per microoperation. Thus if three microoperations are never executed at the same time they may be encoded into 2 bits. The code 00 would specify execution of none of the operations, 01 the execution of the first microoperation, etc. Horizontal microinstructions almost always use field encoding. For example, if there are three inputs to a register, it is generally true that only one input data path would be chosen at a time. Field encoding is also useful for vertical microcoding. Typically there will be an operation field, which will set up the data paths and transformation unit control, and source and destination fields (some use a single field), which set up the register outgating and ingating.

There are two disadvantages in using field encoding. First, the choice of microoperations that are mutually exclusive is not always

clear. Secondly, each stage of decoding of the microoperation will decrease the speed of operation by the delay time of one gate. Thus, for large codes either large delays will be suffered or large amounts of circuitry will be required to decode the microoperation.

A second modification which is used to reduce the word length is word encoding. In word encoding one field of the microinstruction defines the meanings of the other sections of the control word. For example, a section of a horizontal microinstruction might control memory operations if a certain bit of the word were a zero, but if the bit were a 1 the same section that controlled the memory would now control the I/O data paths. In a vertical microinstruction, the operation code field may change meaning depending upon the state of the word encode field. The disadvantage of the use of word encoding is a reduction in the parallelism of the microinstruction.

A number of authors use terms different than horizontal and vertical microprogramming. The more descriptive terms of minimally decoded (horizontal microprogramming) and highly decoded (vertical microprogramming) have been used by Lawson [L3]. The term function field type (horizontal microinstruction) and machine code type, or mini-instruction (vertical microprogramming) have been used by Ramammorthy [R1].

The terms monophase, polyphase, encode, and little encode have been used by Redfield [R2]. The terms encode and little encode correspond to the terms minimumly decoded and highly decoded. The terms monophase and polyphase refer to the duration and sequencing of control signals. A monophase microinstruction produces a simultaneous set of control signals. A polyphase microinstruction produces a set of control signals in which each member of the set may be active at different times during a microcycle time span. In discussions at conferences the terms long word (horizontal microprogramming) or short word (vertical microprogramming are occasionally used. In this paper we will use the traditional terms of horizontal and vertical microprogramming. A further discussion of horizontal and vertical microprogramming will help clarify the difference between the two methods.

Each horizontal microoperation controls gates (either input gates or output ) within the CPU, thereby causing data to flow into or out of registers and transformation units. A register to register transfer is executed by the proper selection of the horizontal microoperations, each executed at the same time. It is the freedom of selecting the data paths, and therefore the register

to register transfers, which gives the horizontal microprogram control method great flexibility.

The flexibility offered by the use of horizontal microprogramming has its price. The length of microinstruction words for a horizontal microprogram controlled machine is longer than for a vertical microprogram controlled machine. Also, the programming of a horizontal microprogram machine is tedious. It is much easier to specify a register to register transfer than to specify all the microoperations required to implement the register to register transfer.

Horizontal microprogramming was used in some models of IBM's System 360 series of computers. It is interesting to note that the horizontal microinstruction word length used in the System 360 series computers varied from 56 bits (model 30) to 100 bits (model 65). A section of a CPU is shown in Figure C.2. The possible microoperations are listed in Figure C.3 and shown (dotted lines) in Figure C.2, Several register to register transfers, together with the microoperations required to implement them are listed in Figure C.4. The examples of Figure C.4 are very straightforward and need no explanation.

Vertical microprogramming employs the individual bits of the microinstruction word to specify register to

register transfers. The register to register transfer must be executable in one microinstruction cycle time. For most CPU's the number of different register to register transfers that may be implemented is very large. Normally, a small subset of all possible transfers is chosen as the set of microoperations. Since a restricted set of transfers is normally available in a vertical microprogrammed machine, it is not as flexible as a horizontal microprogrammed machine. However, the microinstruction word length for vertical microprogramming is normally smaller than the microinstruction word length for horizontal microprogramming. In addition, the vertical microprogram machine is easier to microprogram than a horizontal microprogram machine. Vertical microprogramming is very similar to machine language or assemble language programming of a conventional computer.

An example of a vertical microprogrammed machine is the Interdata 3 computer. The simplified CPU organization and the microoperations for the Interdata 3[I3] are shown in Figures C.5 and Figure C.6, respectively. There are only 16 basic microoperations provided by the Interdata 3. However, the flexibility of the operations is increased by allowing the programmer to specify the destination and source registers independently.

The Interdata Model 3 microinstruction word length is 16 bits.

The smaller word size results in a reduction in the total number of control storage bits required, but also results in a loss in flexibility. For example, it would not be possible to add two registers together and store the result, for further use, in two registers instead of a single register. The data paths are available to accomplish such an operation but because of the restricted control word length, the control bits to specify two result registers instead of a single result register are not available. With horizontal microprogramming such an operation would be possible (see Figure C.4 for an example).

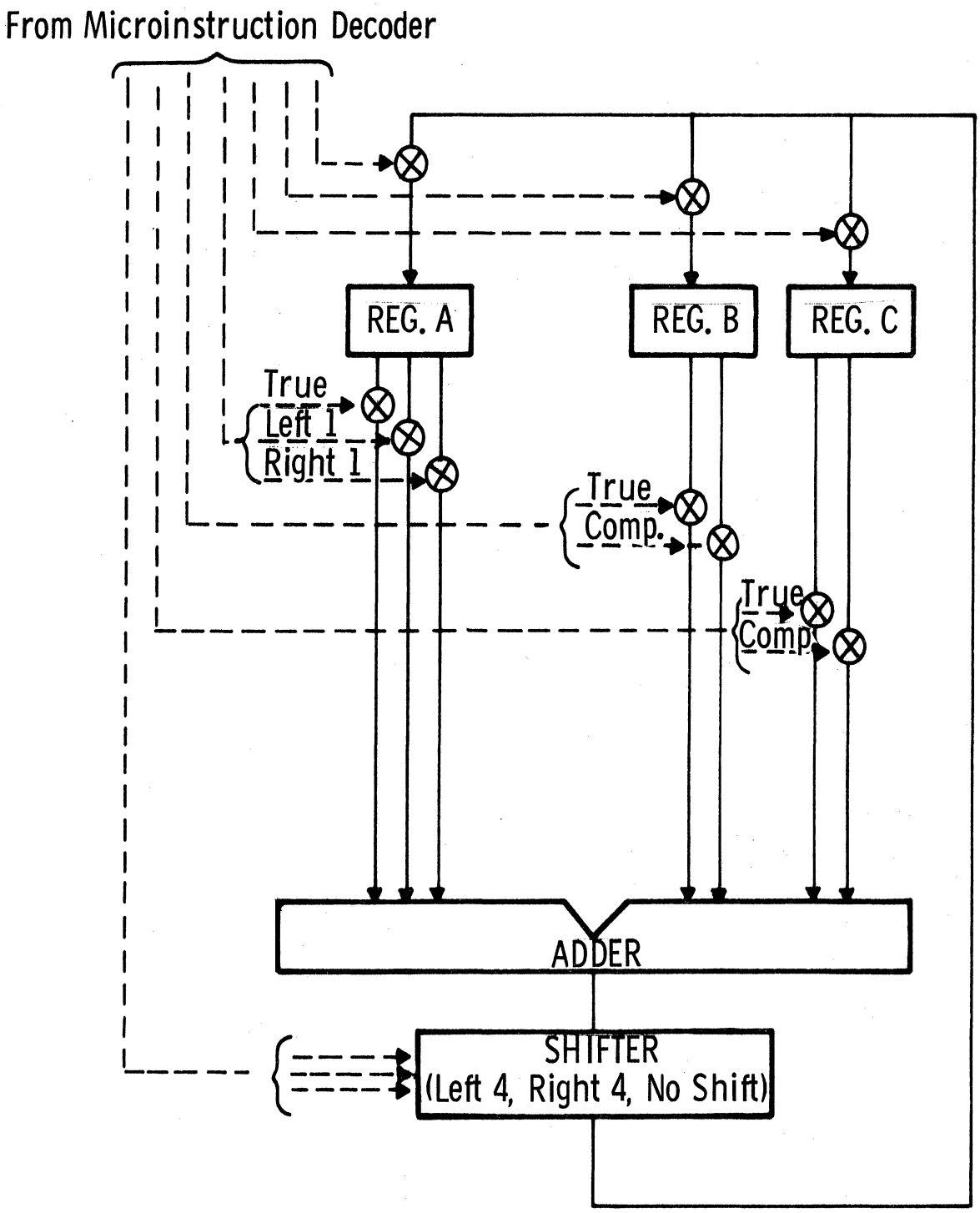
The two types of microprogramming (vertical and horizontal) are very different approaches to CPU control. The use of micro-operations as gate controls (horizontal microprogramming) is essentially the replacement of hardwired control logic by a control memory. The control signals normally provided by a conventional logic control unit are provided by the microoperations of the horizontal micro-instruction word. For example, the microoperations of Figure C.2 and Figure C.3 are identical to the operations that would be listed on a flow diagram used to document hardwired control logic.

The use of microoperations as register to register transfers is essentially the simulation of one computer by another computer. The microoperations of the vertical microprogrammed computer are very basic machine instructions. For example, the IBM 360 Model 25 [I1] microinstructions actually use four basic clock cycles each of

which can cause a register to register transfer. The Interdata Model 3 can accomplish only one register to register transfer per microinstruction. However, in both cases the microinstructions perform the standard machine operations of ADD, SUBTRACT, and TRANSFER. This can be observed from the microoperation of the Interdata 3 (Figure C.6). Because the microoperations are held in a control memory with very fast access times, the microinstructions can be executed very rapidly. To execute an Interdata 3 machine instruction, many Interdata 3 microinstructions have to be performed. Therefore, the Interdata 3 computer can be viewed as being simulated by an elementary computer which is the microcontrol computer. The term "elementary computer" refers to the very basic nature of the microoperations which form the instruction set of the control (microprogrammed) computer.

Before concluding this section one use of the term microprogramming should be mentioned. Microprogramming is sometimes used to describe a form of instruction coding. A good example of this is the PDP-8I microinstruction code which is shown in Figure C.7. The instruction, which is a machine instruction held in main memory not in a control memory, allows the programmer to specify different combinations of CPU operations to be performed during one main memory cycle. A programmer, by placing a 1 in bit position 11

also increments the accumulator (IAC) after it has been cleared. By choosing combinations of the bits of the code 7 instruction word, the programmer forms his own machine instructions. However, this is not microprogramming in the normal sense. The 7 op code is recognized and the instruction implemented with hardware logic control. In fact, the combination of clear the accumulator (CLA) and complement the accumulator (CMA) occurring together must be recognized and treated specially to achieve the desired results.




Section of a CPU

Figure C.2

| Symbol | Function |
|---|---|
| A | Gate Reg. A to Adder Left Side |
| AL1 | Shift Reg. A Left 1 to Adder Left Side |
| AR1 | Shift Reg. A Right 1 to Adder Left Side |
| B | Gate Reg. B to Adder Right Side |
| BC | Gate Reg. B Complemented to Adder Right Side |
| C | Gate Reg. C to Adder Right Side |
| CC | Gate Reg. C Complemented to Adder Right Side |
| NOS | Pass Adder Output through Shifter with no Shift |
| SL4 | Pass Adder Output through Shifter with Left 4 Shift |
| SR4 | Pass Adder Output through Shifter with Right 4 Shift |
| SA | Set Shifter Output into Reg. A |
| SB | Set Shifter Output into Reg. B |
| SC | Set Shifter Output into Reg. C |

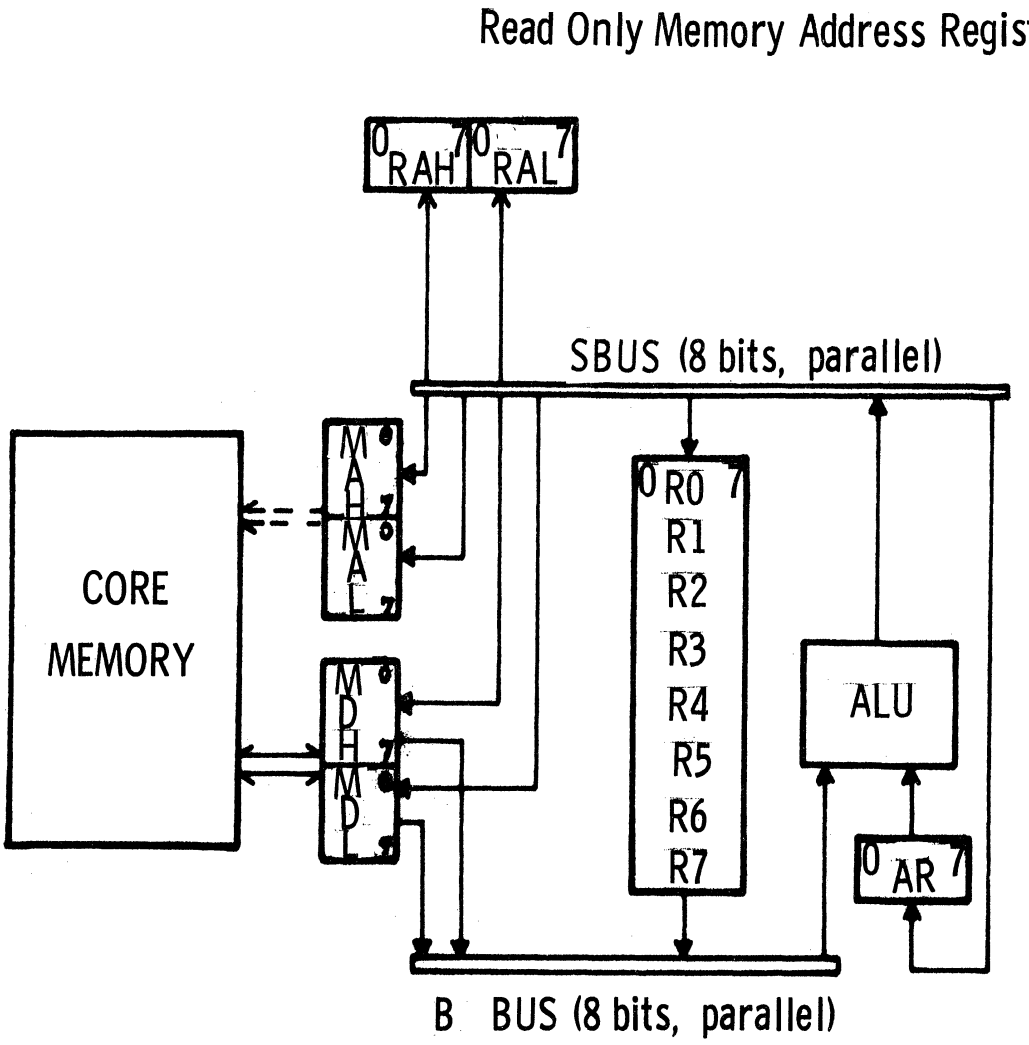
Microoperations for Figure C.2

Figure C.3

| Operation | Microorder |
|---|---|
| Transfer the contents of register A shifted right 4 places into register B<br><br>A(shift right 4) → B | A, SR4, SB |
| Transfer the contents of register A added to the contents of register B into register C<br><br>A + B → C | A, B, NOS, SC |
| Transfer the contents of register A added to the contents of register B into register C and register A<br><br>A + B → C; A + B → A | A, B, NOS, SC, SA |

Some Possible Register to Register Transfers for the CPU of Figure C.2 and the Microoperations of Figure C.3

Figure C.4




Simplified Block Diagram of Interdata Model 3

Figure C.5

| Op Code | Mnemonic | Instruction | Format | execution time(nsec) |
|---|---|---|---|---|
| C | A | Add | A | 760 |
| D | A | Add Immediate | B | 760 |
| E | S | Subtract | A | 760 |
| F | S | Subtract Immediate | B | 760 |
| A | X | Exclusive OR | A | 380 |
| B | X | Exclusive OR Immediate | B | 380 |
| 8 | N | AND | A | 380 |
| 9 | N | AND Immediate | B | 380 |
| 6 | O | Inclusive OR | A | 380 |
| 7 | O | Inclusive OR Immediate | B | 380 |
| 4 | L | Load | A | 380 |
| 5 | L | Load Immediate | B | 380 |
| 3 | C | Command | C | 380 |
| 2 | T | Test | C | 380 |
| 1 | B | Branch on Condition | D | 760 |
| 0 | D | Do | - | - |

Format A –

| Part: | Op | D | S | E |
|---|---|---|---|---|
| Size (bits): | 4 | 4 | 4 | 4 |

Format C –

| Part: | Op | TC |
|---|---|---|
| Size (bits): | 4 | 12 |

Format B – O

| Part: | Op | D | Data |
|---|---|---|---|
| Size (bits): | 4 | 4 | 8 |

Format D –

| Part: | Op | Mask | Address |
|---|---|---|---|
| Size (bits): | 4 | 4 | 4 |

Microoperations Interdata Model 3

Figure C.6

Op - specifies operation to be performed.
D - destination register number.
S - source register number.
E - extended operation field, which specifies options within the same operation.
Data- immediate operand (literal).
TC - test command, which specifies the signal to be tested or the operation to be performed.
Mask - condition code mask (C, V, G, L).
Address - destination address.

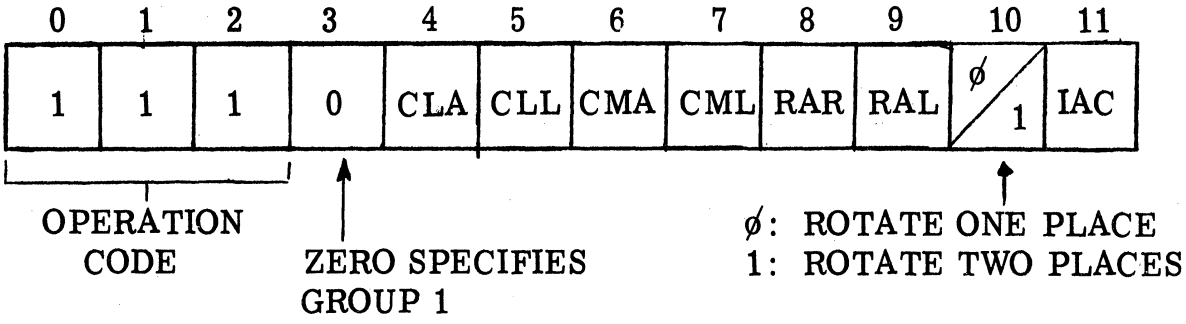
Destination/Source Registers

| Code | Register | Code | Register | Code | Register |
|---|---|---|---|---|---|
| 0000 | R0 | 0101 | R5 | 1010 | MDL |
| 0001 | R1 | 0110 | R6 | 1011 | MAH |
| 0010 | R2 | 0111 | R7 | 1100 | MAL |
| 0011 | R3 | 1000 | AR | 1101 | RAH/SDR |
| 0100 | R4 | 1001 | MDH | 1110 | RAL/SCR |
| | | | | 1111 | DFR |

Microoperations Interdata Model 3

(Cont.)

Figure C.6




CLA - clear accumulation

CLL - clear link

CMA - complement accumulation

CML - complement link

RAR - rotate right

RAL - rotate left

PDP-8 Microinstruction

Figure C.7

# APPENDIX D

## Automatic Generation of B Tables

The design method developed in the body of this report requires the generation of one or more implementation methods for each program type. There is one implementation for the base hardware and one or more improved implementations. Each of the improved implementations uses one or more hardware options. The required hardware options for the implementations of a program type are represented by the $B^i$ matrix and the execution time and storage requirements are represented by the $V^i$ and $S^i$ vectors, respectively. The $B^i$ matrix, the $V^i$ vector, and the $S^i$ vectors can be generated from the base implementation method of a program type. This is the method which would normally be used by the designer. The designer would develop the base implementation method and then note the change in execution time and required storage if the hardware options (individually or in combinations) were added to the base hardware. The procedure of generating the $B^i$ matrix and the $V^i$ and $S^i$ vector from the base implementation method could be partially automated. A discussion of the

semi-automated approach is given below.

The approach would be to identify sequences of microcode in the base implementations which could be improved using the hardware options. A program type and the base microcode for the base hardware of Figure 5.2.1 are shown below (Figure D1). The sequence of microcode of Steps 1, 2 and Steps 5, 6 could be replaced by one microinstruction if the double width ALU option was chosen. Both sequences are an ALU operation performed on both the left and right half of a word of control storage. Such a sequence could be noted by the hardware designer and used in an algorithm for detecting such occurrences in the base hardware implementations.

An interesting problem is illustrated in Figure D1. The microinstructions (1, 2) could be replaced by 1 microinstruction if hardware option 8 (Figure 5.2.2) were chosen, but microinstructions (1, 2, 3) could be replaced by 1 microinstruction if hardware options 4 and 7 (Figure 5.2.2) were chosen. This occurs because microinstructions (1, 2, 3) of Figure D1 implement a test to determine if S2 is less than LENGTH. This test can be performed in one microinstruction if options 4 and 7 (Figure 5.2.2) are added to the base hardware (Figure 5.2.1). Thus the procedure would have to detect both occurrences, and form the three different implementations which result if option 8 or options 4, 7, or options 4, 7, 8 were chosen. This implies that the

designer would probably scan the base hardware program type implementations for occurrences of difference sequences, and then let the automated procedure find the sequences and form the different implementations some of which would result from combining sequences.

There are some sequences which would be very hard to identify For example, the identification of a sequence which was performing a stack operation would be very difficult to identify since it would probably be a complicated procedure and might vary slightly from implementation to implementation. Thus there are some sequences and hence some implementations which would still have to be provided by the designer.

The question of the correctness of the automatically generated implementations would be a problem. That whole question of program correctness is a major research area and will not be discussed here.

```
        S1←S2 TR  ADDR  S3  LABEL
1         ←RB2(LENGTH) + 1 + RB1(S2)      IN BOUNDS TEST
2         ←LB2(LENGTH) + C + LB2(S2)
3          BRA1  8  LABEL
4          RB3(T) ← RB1(S3) + RB2(S2)
5          LB3(T) ← LB1(S3) + C + LB2(S2)
6          MA ← B(T)       MEMC
7          NOP
8          NOP
9          B3(S1) ← MB
10         BR DONE(T + 1)
```

Program Type Implementation

Figure D1

# BIBLIOGRAPHY


[ B1] Barton, N., R. McGuire, "System 360 Model 85 Microdiagnostics," Spring Joint Computer Conference, AFIPS Conference Proceedings, vol. 36, 1972.

[ B2] Bell, C.G., and A. Newell, Computer Structures: Readings and Examples, McGraw-Hill Co., New York, 1970, Chapter 2.

[ B3] Bell, C.G., and A. Newell, "PMS and ISP Descriptive Systems for Computer Structures," AFIPS Conference Proceedings, vol. 36, May 1970, pp. 351-374.

[ B4] Bowman, Sally, T. J. DeSchon, "Case Study on Emulation of the 160A in a Complex Environment," preprints 4th Annual Workshop on Microprogramming, University of California, Santa Cruz, California, Sept. 1971.

[ B5] Breuer, M. A., "General Survey of Design Automation of Digital Computers," IEEE Proceedings, vol. 54, No. 12, Dec. 1966, pp. 1708-1721.

[ B6] Breuer, M.A., "Recent Developments in Automated Design and Analysis of Digital Systems," IEEE Proceedings, vol. 60, No. 1, Jan. 1972, pp. 12-27.

[ C1] Calhoun, R.C., Diagnostics at the Microprogramming Level, Modern Data, May 1969, p. 8.

[ C2] Conway, Melvin E., "Proposal for an UNCOL," Comm. of ACM, vol. 1, No. 10, Oct. 1958, pp. 5-8.

[ C3] Cook, R. W., M. J. Flynn, "System Design of a Dynamic Microprocessor," Department of Electrical Engineering, Northwestern University, Evanston, Illinois, 1969.

[ D1] Darringer, J. A., The Description, Simulation and Automated Implementation of Digital Computer Processors, Ph. D. Dissertation, Carnegie-Mellon University, Pittsburg, Pa., May 1969.

[ D2] Darringer, J. A., "A Language for the Description of Digital Computer Processors," report, Carnegie-Mellon University, Pittsburg, Pa., May 1969.

[ D3] Davies, P. M., "Readings in Microprogramming," IBM Systems Journal, vol. 11, No. 1, 1971, pp. 16-39.

[ D4] Day, W. H. E., "Compiler Assignment of Data Items to Registers," IBM Systems Journal, vol. 9, No. 4, 1970, pp. 281-317.

[ D5] Duley, J.R. and D. L. Dietmeyer, "A Digital Design System Language," IEEE Transaction on Computers, vol. EC-17, No. 9, Sept. 1968, pp. 850-861.

[ D6] Duley, J. R., and D. L. Dietmeyer, "Translation of a DDL Digital System Specification to Boolean Equations," IEEE Transactions on Computers, vol C-18, No. 4, April 1969, pp. 305-318.

[ F1] Feldman J., and D. Gries, "Translator Writing Systems," Communication of the ACM, vol. 11, Feb. 1968, pp. 77-113.

[ F2] Flynn, M. J., M. D. MacLaren, "Microprogramming Revisited," Proc. 22nd Natl. Conf. of the ACM (1967), pp. 457-464.

[ F3] Flynn, M. J., R.F. Rosin, "Microprogramming - An Introduction and a Viewpoint," IEEE Transactions on Computers, vol. C-20, No. 7, July 1971, pp. 727-731.

[ F4] Friedman, T. D., "ALERT - A Program to Compile Logic Designs of New Computers," IEEE Digest on the 1st Annual IEEE Computer Conference, No. 16C51, Aug. 1967, pp. 128-130.

[ F5] Friedman, T. D., "Methods Used in an Automatic Logic Design Generator (ALERT)," IEEE Transactions on Computers, vol. C-18, No. 7, July 1969, pp. 593-613.

[ F6] Gerace, G. B., "Microprogram Control for Computing Systems," IEEE Transactions on Electrical Computers, vol. EC-12, No. 5, December 1963, pp. 733-747.

[G2] Gorman, D. F., and J. P. Anderson, "A Logic Design Translator," AFIPS Conference Proceedings, Fall,1962, pp. 251-261.

[G3] Gorman, D.F., "Systems Level Design Automation: A Progress Report on the Systems Descriptive Language (SLD II)," IEEE Digest of the 1st Annual IEEE Computer Conference, No. 16C51, Aug. 1967, pp. 131-134.

[G4] Grasselle, A., "The Design of Program-Modifiable Micro-programmed Control Units," IRE Trans. on Electrical Computers, vol. EC-11, No. 3, June 1962. pp. 336-339,

[G5] Green, J. "Microprogramming, Emulators, and Programming Languages," Comm. ACM 9, 3 (March 1960), pp. 230-232.

[G6] Guffin, R. M., "Microdiagnostics for the Standard Computer MLP-900 Processor," IEEE Trans. on Computers, vol. C-20, No. 7, July 1971, pp. 803-808.

[H1] Hasset, A., "The Design of a High Level Language Machine," IBM Scientific Center, Palo Alto, California, 1971, p. 32.

[H2] Hasset, A., J. W., Lageschulte, L. E. Lyon, Implementation of a High Level Language Machine," IBM Scientific Center, Palo Alto, California, 1971, p. 45.

[H3] Hawryszkiewycs, Igor, T., "Microprogrammed Control in Problem-Oriented Languages," IEEE Trans. on Electronic Computers, vol. EC-16, October 1967, No. 5, pp. 652-658.

[H4] Hoovind, Robert C., "The Many Faces of Microprogramming," Computer Decisions, vol. 3, No. 9, Sept. 1971, pp. 6-10.

[H5] Husson, Samir, S., Microprogramming Principles and Practices, Prentice-Hall, 1970, p. 613.

[I1] IBM Field Engineering Theory of Operation, 2025 Processing Unit, Y24-3527-0, Oct. 1968.

[I2] Interdata Model 3 Microinstruction Reference Manual, Publication Number 29-017R01, April 1968.

[I3] Iverson, Kenneth E. A Programming Language, John Wiley and Sons, Inc., 1962, p. 286.

[J1] Johnson, A. M., "The Microdiagnostics for IBM 360 Model 30," IEEE Trans. on Computers, vol. C-20, No. 7, July 1971, pp. 798-803.

[K1] Kleir, R. L., C. V. Ramamoorthy, "Optimization Strategies for Microprograms," IEEE Trans. on Computers, vol. C-20, No. 7, July 1971, pp. 783-794.

[K2] Knuth, Donald E., "An Empirical Study of Fortran Programs," Computer Science Department Report No. CS-186, Stanford Artificial Intelligence Project Memo AIM-137, Computer Science Dept., Stanford University, 1970, p. 42.

[L1] Lawler, E. L., M. D. Bell, "A Method for Solving Discrete Optimization Problems," Operation Research, vol. 14, No. 6, Nov-Dec 1969, pp. 1098-1112.

[L2] Lawsen, H. W., "Programming-language-oriented Instruction Streams, IEEE Trans., C-17 (1968), pp. 476.

[L3] Lawsen, Jr., H. W., B. K. Smith, "Functional Characteristics of a Multilingual Processor," IEEE Trans. on Computers, vol. C-20, No. 7, July 1971, pp. 732-742.

[L4] Levy, Leon. S., "State of the Art of Microprogramming," Aerospace Corporation, El Segundo, California, April 1965, Contract No. AF 04(695)-469.

[L5] Lowry, Edward S., C. W. Medlock, "Object Code Optimization," Comm. of ACM, vol. 12, No. 1, January 1969, pp. 13-22.

[M1] McBee, W. C., "The TRW-133 Computer," Datamation 10/2, February 1964, pp. 27-29.

[M2] McCormick, M. A., T. T. Schansman, K. K. Womack, "1401 Compatability Feature on the IBM System 1360 Model 30," Comm. ACM, 8, 12 (December 1965), pp. 773-776.

[M3] Metze, G., and S. Seshu, "Proposal for a Computer Compiler," AFIPS Conference Proceedings, Spring 1966,

[P1] Proctor, R., "Logic Design Translator Experiment Demonstrating Relationship of Language to Systems and Logic Design," IEEE Transactions on Computers, vol. Ec-13, August 1964, pp. 422-430.

[R1] Ramamoorthy, C. V., M. Tsuchiyal, "A Study of User-Microprogrammable Computers," Proc. AFIPS, vol. 35, 1970, pp. 165-181.

[R2] Redfield, S. R., "A Study in Microprogrammed Processors: A Medium Sized Microprogrammed Processor," IEEE Trans. on Computers, vol. C-20, No. 7, July 1971, pp. 743-750.

[R3] Rosin, Robert F., "Contemporary Concepts of Microprogramming and Emulation," Computing Surveys, vol. 1, No. 4, (December 1969), pp. 197-212.

[R4] Rosin, Robert F., Gideon Frieder, Richard Eckhouse, Jr., "An Environment for Researdh in Microprogramming and Emulation," preprints (ACM) 4th Annual Workshop on Microprogramming, University of California, Santa Cruz, California, Sept. 1971.

[R5] Rosin, Robert F., "Notes on Emulations and Microprogramming," Advanced Topics in Systems Programming, Engineering Summer Conference, University of Michigan, 1970, p. 62.

[S1] Sammet, Jean E., Programming Languages:, Prentice-Hall, 1969, p. 785.

[S2] Schlaeppi, H. P., "Formal Language for Describing Machine Logic, Timing, and Sequencing (LOTIS)," IEEE Transactions on Computers, vol. EC-13, August 1964, pp. 439-448.

[S3] Schiller, W. L., R. L. Abraham, R. M. Fox, A. van Dam, "A Microprogrammed Intelligent Graphics Terminal," IEEE Trans. on Computers, vol. C-20, No. 7, July 1971, pp. 775-782.

[S4] Schoen, T. A., M. R. Belsole, Jr., "A Burroughs 220 Emulation for the IMB 360/25," IEEE Trans. on Computers, vol. C-20, No. 7, July 1971, pp. 795-798.

[S5] Semarne, H. M., and R. E. Porter, "A Stored Logic Computer," Datamation, 7/5, May 1961, pp. 33-36.

[S6] Shorr, H., "Computer Aided Digital System Design and Analysis Using a Register Transfer Language," IEEE Transactions on Computers, vol. EC-13, Dec. 1964, pp. 730-737.

[S7] Shriver, D. B., Sr., "Microprogramming and Numerical Analysis," IEEE Trans. on Computers, vol. C-20, No. 7, July 1971, pp. 808-811.

[S8] Slobodynonyck, T. F., "A Method of Minimizing Microprograms," An UKrSSR. Kibermeticheskaya Tekhnika, (Academy of Sciences og the Ukrainen SSR. Sybernitecs Tech. Kiev, 1965, pp. 72-80 - Available AD 662-821 Foreign Technology Division, Wright-Patterson AFB, Ohio.

[S9] Steel, T. B., "A First Version of UNCOL," Proceedings of the Western Joint Computer Conference, 1961, pp. 371-378.

[S10] Strong, J., et al, "The Problem of Programming Communications with Changing Machines, A Proposed Solution, Part I," Comm. of ACM, vol. 1, No. 8, August 1958, pp. 8-12.

[S11] Strong, J., et al, "The Problem of Programming Communications with Changing Machines, A Proposed Solution, Part II," Comm. of ACM, vol 1, No. 9, Sept. 1958, pp. 9-15.

[T1] Tucker, S. G., "Emulation of Large Systems," Comm. ACM, 8, 12 (December 1965), pp. 753-761.

[T2] Tucker, S. G., "Micro-program Control for System/360," IBM Systems Journal, 6 (1967), p. 222.

[T3] Tucker, Allen B., Michael J. Flynn, "Dynamic Microprogramming: Processor Organization and Programming," Comm. of ACM, vol. 14, No. 4, 1971, pp. 240-250.

[W1] Weber, H. A., "Microprogrammed Implementation of EULER on IBM System/360, Model 30," Comm. of ACM, vol. 10, pp. 549-558.

[W2] Wilkes, M. V., "Microprogramming," Proc. AFIPS, vol. 12, EJCC, 1958, pp. 18-19.

[W3] Wilkes, M. V., "The Growth of Interest in Mirco-programming, A Literature Survey," Computing Surveys, "vol. 1, No. 3, (September 1963, pp. 139-145.