ASPECTS OF MICROPROGRAMMING: THE UNDERLYING MACHINE STRUCTURE OF MICROPROGRAMMABLE PROCESSORS

by

Michael Allen Houghtaling

A Thesis Submitted to the Faculty of the DEPARTMENT OF COMPUTER SCIENCE
In Partial Fulfillment of the Requirements For the Degree of MASTER OF SCIENCE
In the Graduate College THE UNIVERSITY OF ARIZONA

1975
STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Michael A. Houghtaling

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

ROBERT L. BASFORD 8/11/75
Associate Professor of Date
Computer Science
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF ILLUSTRATIONS</th>
<th>iv</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>v</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>vi</td>
</tr>
</tbody>
</table>

## CHAPTER

1. THE PHILOSOPHICAL DEVELOPMENT OF MICROPROGRAMMING  
   - Introduction and Historical Perspective  1
   - Theoretical Foundations  10
   - Definitions  22

2. THE ORGANIZATION AND DESIGN OF A MICROPROGRAMMABLE CONTROL ENGINE  35
   - A Microprocessor Model  35
   - Design Strategies  42

3. ON THE CONCEPTUALIZATION OF MACHINE STRUCTURE  64
   - The Hardware-Firmware-Software Mix and Microprogramming Applications  64
   - Programming Languages for the Specification of Microprograms  78
   - Microprocessor Architecture and Implications  85
   - Conclusions  92

SELECTED BIBLIOGRAPHY  96
# LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Illustration</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Wilkes' Microprocessor</td>
<td>6</td>
</tr>
<tr>
<td>2.</td>
<td>Glushkov's Automata Model</td>
<td>13</td>
</tr>
<tr>
<td>3.</td>
<td>Subautomata of Control Automaton</td>
<td>20</td>
</tr>
<tr>
<td>4.</td>
<td>Microprogrammable Processor Model</td>
<td>38</td>
</tr>
<tr>
<td>5.</td>
<td>Conventional Processing Unit</td>
<td>38</td>
</tr>
<tr>
<td>6.</td>
<td>Register--Bus Data Paths of Model CPU Configuration</td>
<td>40</td>
</tr>
<tr>
<td>7.</td>
<td>Microword Formats</td>
<td>57</td>
</tr>
<tr>
<td>8.</td>
<td>Horizontal Microinstruction Format for Model Microprocessor</td>
<td>59</td>
</tr>
<tr>
<td>9.</td>
<td>Vertical Microinstruction Format for Model Microprocessor</td>
<td>60</td>
</tr>
<tr>
<td>10.</td>
<td>Iterative Tuning Process</td>
<td>77</td>
</tr>
<tr>
<td>11.</td>
<td>High-level Microlanguage Compiler for Generation of Efficient Horizontal Microcode</td>
<td>83</td>
</tr>
<tr>
<td>12.</td>
<td>Microprocessor System Configuration</td>
<td>88</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Design Strategies and Implications for Microinstruction Formats</td>
<td>62</td>
</tr>
<tr>
<td>2. Characteristics of Horizontal and Vertical Microprocessors</td>
<td>63</td>
</tr>
</tbody>
</table>
ABSTRACT

This paper presents a characterization of the domain of microprogramming and the philosophical consequences of its integration into the structure of virtual system design. The explication begins with a review of the historical and theoretical foundations of microprogramming concepts. Subsequent exposition on microprogramming as a design technique and medium for system realization is organized around three conceptual aspects: the physical organization of microprogrammable hardware, the process of microprogram specification, and the types of algorithms which are conducive to microprogrammed implementations. Interrelationships between design decisions which occur within each category are discussed. System resources are partitioned into two functional components, the operating engine and the control engine. It is shown that the structure of the operating engine is dependent upon virtual system applications while mechanisms necessary for control of the operating engine will determine the structure of the control engine. Design strategies for development of a microinstruction format which provides for the efficient manipulation and management of these primitive resources are explained in a taxonomy of microinstruction formats. Implications concerning the hardware-firmware-software mix and boundary determination of future systems are examined and some areas for further research are mentioned.
CHAPTER 1

THE PHILOSOPHICAL DEVELOPMENT OF MICROPROGRAMMING

Introduction and Historical Perspective

The possibility of a third resource has been added to the two traditional constituents of digital computers. Realization of virtual systems may now be distributed among hardware, firmware, and software media. Numerous proposals for microprogrammed solutions to the complex problems involved in the design and organization of efficient but flexible machine architectures have produced an extensive growth of literature on the subject of microprogramming.

This paper presents a characterization of the domain of microprogramming and a discussion of the philosophical consequences which result from its integration into virtual system structure. A study of much of the literature was undertaken in order to gain some consensus on the many approaches to microprogrammed implementations described by the articles. A synthesis of the conceptual basis and design strategies implied by such applications is provided. The many aspects of microprogramming are grouped according to principles which are concerned with the nature of microprogrammable hardware, the attributes of microprogramming languages, and possible microprogramming applications. An examination of the interrelationships between these principles, with the aim of
formulating directions and questions for further research concludes the paper.

The traditional approach towards the design and implementation of computer systems involved a well-defined partition between hardware and software. Digital computer engineers were assigned the task of designing hardware that would accept some appropriate machine language. The set of individual machine instructions within the machine language represented a computer's basic vocabulary. This vocabulary incorporated the elementary functions and operations available for computational purposes. Specific machine functions included in the hardware design were selected in accordance with the objectives and purposes desired as system goals, either for a specialized or general-purpose architecture.

The selected functional tasks would be expressed in terms of a machine language instruction set and realized by permanently wired combinational logic and sequential circuits. The irrevocable wiring of hardware necessitated considerable forethought and deliberation during initial system specification and design. After the hardware was physically realized, it became the task of system or utility programmers (software engineers) to provide software which would allow the system to respond to directives expressed in terms of procedure-oriented languages. The logical and physical distinctions between the two groups of designers responsible for the over-all system development tended to isolate each group within its own sphere of influence.

The partitioned approach for implementation of digital systems, with its rigid distinction between hardware and software development,
contributed to the growth of detrimental consequences for users of
general-purpose computers. The systems generated by the partitioned
design philosophy were not efficiently, economically or easily applied
towards various computational and general problem-solving tasks. The
translation of programs written in high-level, procedure-oriented
languages into equivalent machine language programs requires that a
certain percent of the computer's time be expended in the transforma-
tional process. This is time in which the computer is not directly
involved in the production of answers to the problems submitted by users.
From a user's point of view, this may be considered a waste of time and
money since he must pay for computational resources which do not directly
produce results (Bashkow, Sasson and Kronfeld 1967, p. 485).

It eventually became apparent that the architectural components
of general-purpose hardware and the inherent structure of machine
languages did not provide appropriate mechanisms for the efficient reali-
zation of applications-oriented programming tasks (Church 1970; Lawson
1968; Reigel, Faber and Fisher 1972; Rosin 1969; Van der Poel 1962;
Wilner 1972). Contortions of data structures and instructions into
unnatural fields induced by the rigid information containers of machines
(memory cells and registers) produced inefficient and complicated mappings
of high-level languages into machine languages (Wilner 1972). While the
classical Von Neumann machine design was constructed for the sequential
execution of single address instructions, the topology of high-level
language programs is more closely related to such nonlinear structures as
pushdown stacks, trees and lists (Haavind 1971, p. 14). This lack of
structural isomorphism between programming languages and general-purpose hardware, and the resultant complex, inefficient mapping functions, demonstrated that the development of programming-language-oriented machines and instruction sets should be initiated (Lawson 1968, p. 476).

Further design complications resulted for both hardware and software engineers as the fields of endeavor in which the assistance of computers was sought increased in breadth and complexity. Although the theory of machine logic and switching circuits provided considerable assistance to hardware engineers, the immense size of the sequential circuits required to control large digital systems prevented the application of systematic design techniques.

The introduction of a systematic design tool was proposed by M. V. Wilkes in 1951. Wilkes offered his proposal as an alternative to the usual ad hoc or "black art" methods employed in the design of logic circuits for the control of hardware facilities (Wilkes 1951, 1969; Haavind 1971). The concept of machine design proposed by Wilkes became known as microprogramming. It was based on the realization of the exponential nature of machine operations; the fact that all possible machine instructions could be built up through the sequential (and in some cases simultaneous) manipulations of the processor's elementary functional units (Mercer 1957, p. 157). Wilkes viewed the execution of a machine instruction as a sequence of transfers of information between processor registers. He noted the analogy between the execution of these individual steps and the execution of individual instructions in
a program (Wilkes 1969, p. 139). The set of individual micro-operations\(^1\) associated with each machine instruction would compose a microprogram. Wilkes suggested that the orchestration of the control signals to the hardware facilities necessary to implement a machine instruction be embodied in a small fast memory, rather than in the usual method of combinational logic and sequential circuits (Haavind 1971, p. 10). The desired set of commands to the hardware could be selected by addressing a microinstruction sequence in control memory, possibly by means of the operation code of the machine instruction.

The actual realization of this concept, as depicted in Figure 1, involved the proposal to utilize a read-only memory consisting of two diode matrices; a "connection matrix" C and a "sequencing matrix" S. (Mercer 1957, p. 158; Wilkes 1969, p. 139). The two matrices correspond to two fields in the microinstruction, one for specifying the functional operations to be executed and one for selecting the next microinstruction to be executed. The connection matrix is used to specify the appropriate gate connections for transfer of information to arithmetic-logic units and control registers. Each vertical line in matrix C corresponds to a micro-operation; activation of a vertical line accomplishes the particular register transfer or arithmetic function. Each horizontal line corresponds to one micro-order\(^2\) or set of micro-operations that are executed simultaneously (Mercer 1957, p. 158). The connections in

1. Definition for the term "micro-operation" is provided on page 23.

2. Definition for the term "micro-order" is provided on page 23.
Figure 1. Wilkes' Microprocessor.
matrix C indicate points at which a pulse on the micro-order line is transferred to the proper vertical lines for execution of the micro-operations that make up a given micro-order. The address of the next microinstruction is determined by the sequence matrix 'S', setting its number into the operation or 'O' register. The bit settings of the 'O' register serve as inputs to the decoding tree which routes the incoming clock pulse to the correct micro-order line of the two matrices. Switches or flip-flops are used at points 'X' and 'Y' to allow conditional decisions within the machine. Either of two micro-orders can be selected for execution by decision element 'X' and either of two micro-orders can be selected as the next microinstruction by decision element 'Y'. (Mercer 1957, p. 158; Wilkes 1969, p. 139).

The application of the microprogramming concept as a systematic design tool to assist in the construction of digital computer control units was the principal motivation for the proposal by Wilkes. But this orderly design procedure possessed a side effect of significant consequence; it permitted a form of programming one level beneath traditional machine language programming. This "deep structure" perspective of microprogramming provided new forms of underlying machine structure conceptualization. During the subsequent two decades after the introduction and explication of the concept by Wilkes, microprogramming applications were retained in the rather limited sphere of the hardware design engineers. But the gathering forces of technological innovations and economic pressures gradually made the full potentiality of microprogramming accessible (Bell et al. 1970, p. 658; Haavind 1971, p. 10; Davidow 1972a, Part 1; Barr et al. 1973, p. 976; Reyling 1974, p. 81).
Investigation into the means of reducing the mis-matched topological structure between machines and programming languages, and research into techniques for increasing system throughput and efficiency produced tantalizing possibilities for microprogrammed solutions to these and other problems (Opler 1967; Weber 1967; Flynn and MacLaren 1967; Tucker and Flynn 1971; Rosin, Frieder and Eckhouse 1972; Liskov 1972; Wilner 1972; Davis, Zucker and Campbell 1972; Reigel, Faber and Fisher 1972; Cook, Sisson, Storey and Toy 1973). A synthesis of the conceptual basis for such proposals is attempted in this paper.

The domain of microprogramming, its characterization, and its influence and integration as a technique for the design and development of digital computer systems will be considered in the development of this thesis. A considerable amount of literature has been published on the various aspects of microprogramming. The nature and characteristics of the concept, together with particular instances of implementation are central themes of discussion that frequently occur. Some development on the theory of microprogramming has also been presented (Glushkov 1964, 1965; Ito 1973). In order to identify the theoretical and applicative foundations of microprogramming, a study of this body of knowledge was undertaken.

Explications on the nature of microprogramming may be usefully organized into three conceptual levels: (1) microprogrammable computer architecture (hardware), (2) the process of microprogramming and facilities for such activity (software), and (3) firmware applications to computational systems and algorithms. The first level is concerned with
the organization of machine architecture, the influences of technology, and the strategies of system design that support and allow construction of microprogrammable hardware. Once such hardware resources become available, one becomes concerned with the process of generating microcode. At this second level, the traditional aspects of programming language theory and practice are applied towards the development of microassemblers and high-level support languages for the production of microprograms. The third conceptual level in the domain of microprogramming involves the study of the manner in which the techniques of microprogramming can be applied to appropriate tasks throughout the hierarchical structures of computational systems. Examples of microprogramming applications often cited in the literature include emulation, error detection, diagnostics, memory management, floating point arithmetic, address generation, stack operations, sorting, interrupt processing, format checking, compilation, trigonometric functions, matrix operations and fast Fourier transforms (Davis 1974, Falk 1974). Microprogrammed implementation of programming languages is another application area that has received much emphasis.

All three levels are interdependent; the nature of the application influences the design of suitable hardware which in turn determines the characteristics of appropriate support languages, or vice versa. An examination of these relationships is one of the subjects of this paper. This chapter continues with a description of theoretical foundations and definitional terms which are common to the subject matter. Chapter 2 presents an explanatory microprocessor model and a discussion
of design strategies for microprocessor architecture. Chapter 3 expounds on the relationships between the three conceptual partitions of microprogramming, investigates the determination of appropriate hardware-firmware-software mixes in system structure and speculates on probable consequences for machine conceptualization and organization.

Theoretical Foundations

The influence of automata theory has been a principle factor in the development of a formal theory of microprogramming (Berndt 1970; Gerace et al. 1971; Glushkov 1964, 1965; Ito 1973; Stabler 1970; Kleir and Ramamoorthy 1971). Glushkov constructed a formal mathematical apparatus which allowed the application of abstract automata theory to the formalization of microprogramming (Glushkov 1965, p. 1). A formal specification of a microprogrammed computer structure was presented by Ito as he further extended and developed Glushkov's model (Ito 1973, pp. 6-7). It was anticipated that the establishment of a formalized theory of microprogramming would be useful in the logical design and synthesis of computers (Ito 1973, p. 7; Glushkov 1965, p. 1). Construction and development of a theory could produce insight into the nature of machine structure. Generalizations and abstractions derived from the theory could be applied to problems encountered in the system design process. Specifically, the problems involved with the selection of appropriate sets of micro-operations, the construction of microprograms, the transformation and minimization of microprograms, the decomposition of a computer through microprogram transformations, and the verification of
microprogram correctness were topics thought to be amenable to a theoretical approach (Ito 1973, p. 7).

While the considerations of Glushkov and Ito were directed primarily towards the specification of a theory for these purposes, their mathematical models will be discussed in this section as a means of introducing the theoretical concepts underlying the philosophy of microprogramming. Explication of these concepts leads quite naturally into a presentation of standard definitions encountered in the literature.

Automatons are the primitive objects employed in the development of mathematical models for microprogrammed computers. For purposes of familiarization, a brief descriptive review on the formal definition of automata is provided. For further reference see the more complete and detailed formulation of A. W. Burks and Hao Wang (1957). The concepts of automata were developed in order to study objects or systems which changed their state in time. In order to describe the state of such an object at any arbitrary time it is necessary to consider the time instant, the past history, the state of the object's environment, and the laws governing the object's inner action and its environmental interaction (Burks and Wang 1957, pp. 193-194). An object or system of this type may or may not change its size in time and it may or may not interact with its environment. Entities of this nature can be referred to as automatons and could represent a physical body, a machine, an animal or a solar system. Automata are essentially information-transforming systems (Nelson 1969, p. 428). We are particularly concerned with
finite systems and objects. Burks and Wang 1957, p. 201) formally define a finite automaton in the following manner.

A (finite) automaton is a fixed finite structure with a fixed finite number of input junctions and a fixed finite number of internal junctions such that (1) each junction is capable of two states, (2) the states of the input junctions at every moment are arbitrary, (3) the states of the internal junctions at time zero are distinguished, (4) the internal state (i.e., the states of the internal junctions) at the time t + 1 is completely determined by the states of all junctions at time t and the input junctions at time t, according to an arbitrary pre-assigned law (which is embodied in the given structure). ¹

The operation of a particular automaton can be essentially characterized by two transformations (δ and λ) from pairs of integers to integers. The integers are drawn from the finite sets [I], [S] and [O] where I is the set of input symbols, S is the set of internal states and O is the set of output symbols. The transformations are formally specified by Burks and Wang (1957, p. 203) as

\[ S(0) = S_o \] (the distinguished integer)

\[ S(t + 1) = \delta(I(t), S(t)) \]

\[ O(t) = \lambda(I(t), S(t)). \]

The δ transformation represents the state transition function while λ refers to the output function.

Glushkov's mathematical model represents the computer as a composition of two automata, an operational automaton and a control automaton. The relationship between the two automata, as depicted in Figure 2, is one of input/output information exchange. The output of the control

¹ Burks and Wang use time t + 1 at this point in their definition, but to be consistent with their description on page 203 it should be time t.
C = \{P,F,S,M,A\} \text{ where }

\begin{align*}
P &= \text{Input Set of Predicates Reflecting Status Information.} \\
F &= \text{Output Set of Control Signals to Trigger Execution of Micro-operations (f_i).} \\
S &= \text{Set of Internal Control States.} \\
M &= \text{State Transition Function:} \\
\quad P \times S &\rightarrow S \\
\quad S(t+1) = M[S(t),P(t)] \\
A &= \text{Output Function:} \\
\quad P \times S &\rightarrow F \\
\quad F_i(t) = A[S(t),P(t)]
\end{align*}

O = \{V,Z,U,\pi,\varphi\} \text{ where }

\begin{align*}
V &= \text{Input Set of Predicates Reflecting Status Information.} \\
Z &= \text{Output Set of Predicates Reflecting Status Information.} \\
U &= \text{Set of Internal States.} \\
\pi &= \text{State Transition Function:} \\
\quad V \times U &\rightarrow U \\
\quad U(t+1) = \pi[U(t),V(t)] \\
\varphi &= \text{Output Function:} \\
\quad Z(t) = \varphi[U(t)]
\end{align*}

Figure 2. Glushkov's Automata Model.
automaton serves as input to the operational automaton and output from the operational automaton is taken as input to the control automaton. In addition to its communication with the control automaton, the operational automaton also accepts external inputs and generates external outputs.

The string of values \((f_1, f_2, \ldots, f_n)\) transmitted from the control automaton to the operational automaton represents the command signals which cause data transformations to be executed by the operational automaton. The operational automaton is characterized as a finite Moore machine with a vast number of internal states (Glushkov 1965, p. 1; Stabler 1970, p. 908; Gerace et al. 1971, p. 840). It is capable of performing a finite number of data manipulations and transformations. Either one or several primitive operations are usually capable of being executed during each phase of the machine's basic clock cycle. These elementary operations are referred to as micro-operations. The sequence of command signals \((f_1, f_2, \ldots, f_n)\) received by the operational automaton determines the sequence of the micro-operations to be performed. The execution of a micro-operation corresponds to a mapping of the set \(S\) of internal states into itself. The information provided by the operational automaton to the control automaton consists of logical conditions \((p_1, p_2, \ldots, p_n)\). For any particular current state \(s \in S\), the logical condition \(p_i\) is either true or false. The value of specific logical conditions for each operation cycle depends only on the state of the operational automaton and not on its input signal. The sets of
micro-operations and logical conditions for any particular operational automaton are finite (Glushkov 1965, p. 1).

The control automaton can be characterized as either a finite Moore or Mealy automaton, with a smaller number of internal states (when compared to the operational automaton). Usually, the control automaton is viewed as a Mealy machine in which its output depends on the current input and internal state (Stabler 1970, p. 908; Gerace et al. 1971, p. 840). The control automaton's function is to generate a sequence of command signals which determines the execution sequence of micro-operations to be accomplished by the operational automaton. The logical condition signals received as input to the control automaton are used to affect the generation of particular control sequences.

At this point in the explanation, Glushkov's model can be used to represent either a conventional computer or a microprogrammed computer. The control unit for both types of computers has as its function the direction of data flow activities. Under supervision of the control unit, requisite register transfers and data transformations are triggered by gating selected clock pulses on appropriate control lines at particular time instants (Hill and Peterson 1973, p. 172). Primitive to the structure of a control unit are the notions of a state, a state transition, and output as a function of the present state and input (Stabler 1970, p. 910). Implementation of the control task involves the establishment of the current control state and determination of the next sequential control state (Davies 1972, p. 20). It is the manner in which
the control automaton is realized which distinguishes between conventional computer design and a microprogrammed computer.

In conventional systems, the generation of all necessary control signals is based primarily on a hardwired interpretation of the instruction currently being executed (Dollhoff 1973, p. 92; Hill and Peterson 1973, Ch. 7). The instruction under interpretation is placed in an instruction register which drives a decoder. As the decoder separates the control bits contained in the instruction operation code, the control sequencer produces a time series of pulses which initiate data transfers among the internal components of the processor (Roberts 1969, p. 148).

The important aspect of conventional machines is the fixed nature of the control sequencer. The sequential circuitry and combinational logic required for the implementation of the control functions are hardwired realizations.

Microprogramming provides an alternative technique for the mechanization of the control automaton. Sets of control signals are organized on a word basis and stored in a fixed or dynamically changeable control memory (Husson 1970, p. 20). The bits of control memory are used to replace the gates of random control logic (Davidow 1972a, Part 1, p. 76). The patterns of ones and zeros in the control word represent the states of the control automaton. There usually is a one-to-one correspondence between a microinstruction and a control memory word. A microinstruction is composed of sets of micro-operations that can be executed during the time period of microinstruction interpretation. Individual microinstructions are grouped together to form microprograms.
Particular sequences of command signals determine the functions and operations which are defined for a specific processor. Execution of microprograms by the control automaton causes successive changes in the states of the operational automaton.

Under the microprogramming philosophy, orchestration of the necessary control signals becomes a programming task similar to conventional software development. There is a distinction between the two forms of programming however. Whereas programming in the usual sense refers to the selection of correct order codes for the performance of a desired computation, either from a high level language instruction set or from a machine language instruction set, microprogramming involves the selection of microinstructions for the purpose of control signal propagation. Historically, this technique was introduced as a means of providing an orderly method for the design, organization and implementation of the machine instruction set (Husson 1970, p. 19; Reigel, Faber and Fisher 1972, p. 705). It has since been noted that the programmability of the control unit, especially if the control memory is writable, makes it possible to code an entire application program at the microcode level (Roberts 1969, p. 149; Microprogramming Handbook 1971, pp. 48-52). The implication and consequences of this realization will be discussed at a later point.

The microprogramming approach results in a control unit that is primarily embodied in a memory rather than by a large network of sequential circuits (Hill and Peterson 1973, p. 228). This control memory can be viewed as an array of combinational logic which drives a decoding
network and specifies the output and next state of the control unit as a function of the present state (Hill and Peterson 1973, p. 235). The generation of bit patterns necessary for the realization of desired control functions becomes the responsibility and task of the microprogrammer (Reigel, Faber and Fisher 1972, p. 711). The similarity between the task of microprogram construction and that of conventional software programming was previously mentioned.

The set of microprograms that can reside in control store effectively defines the operation of the control automaton. Clearly, the domain of possible definitions made available through the flexibility of microprogramming is much larger than the domain of a control automaton realized by means of a hardwired control sequencer.

Glushkov proposes further decomposition in the theoretic structure of the control automaton. He notes that any individual microprogram \( M = A_1A_2 \ldots A_n \) can be represented in the form of an automaton (Glushkov 1964, pp. 1-2). The states of the automaton are identified with the instructions of the microprogram. The state transition function is assumed to be sequential, transferring the automaton from state \( A_i \) to the next state \( A_i + 1 \). Branch type micro-operations may alter this sequential transition function however. When microinstruction \( A_i \) contains a branch micro-operation, the actual execution of that micro-operation determines the next state (Glushkov 1964, p. 2). Once a microprogram has been represented by an automaton, the minimization procedures of abstract automata theory can be applied to the automaton. Minimization of the number of states in the automaton also minimizes the
microprogram represented by it (Glushkov 1964, p. 2). Of course, if the length of microprogram entries in the control memory can be decreased without affecting the computational results, increases in economy, efficiency and speed can be obtained.

A synthesis of the mathematical models proposed by Glushkov and Ito is illustrated in Figure 3. The control automaton of Figure 2 is decomposed into two automata, automaton A and automaton B. Automaton A represents the control memory of a microprogrammable processor and is related to the set of internal control states (S) of the original control automaton in Figure 2. The control memory automaton can in turn be considered as a network of automata wherein each individual microprogram that resides within the control store is represented by an automaton. Automaton B is the microinstruction decoder/microsequencer and embodies the state transition function and output function of the original control automaton. The circuitry required in the implementation of automaton B will represent the minimum amount of hardware necessary for a control unit configuration. Usually, the number of components involved will be less than the number of elements required by a conventional hardwired control sequencer (Hill and Peterson 1973, pp. 228-230; Davidow 1972a, Part 1, pp. 76-79; Waldecker 1970). The physical organization of control and operational automata will be subsequently referred to as control and operating engines.

This section has characterized the underlying structure of a microprogrammable processor as a hierarchical system of automatons. The decomposition partitioned what has been traditionally referred to as the
Figure 3. Subautomata of Control Automaton.

\[ A = [I_{\mu_i}, F_{\mu_i}, Sa, \Theta, \Delta] \]

- \( I_{\mu_i} = P_{\mu_i} \cap MAR \); the input set of micropredicates (\( P_{\mu_i} \)) and control memory addresses
- \( F_{\mu_i} = \text{Output Set of external Microinstruction Word Bit Patterns} \)
- \( Sa = \text{Set of Internal Control Memory States (Addresses of Microinstructions)} \)
- \( \Delta = \text{Output Function; } F_{\mu_i}(t) = \Delta[Sa(t)] \)
- \( \Theta = \text{State Transition Function: If } F_{\mu_i} \text{ is an internal microinstruction then next state (microinstruction address) is determined as a result of actual execution of that microinstruction; } \)
- \[ Sa(t+1) = [EVAL(F_{\mu_i}(t)), P_{\mu_i}(t)] \]
- Else the next state (microinstruction address) is the address of the next sequential microinstruction in control memory;
- \[ Sa(t+1) = Sa_i(t) + 1 \]
External microinstructions result in information which changes the state of the operational automaton.

Internal microinstructions affect only the state of the control automaton.

\[ B = [V, W, Sb, \Omega, \Sigma] \]

\[ V = P \cup F \] ; the input set of predicates and external microinstruction bit patterns

\[ W = F \cup P \] ; the output set of control signals and micropredicates.

\[ Sb = \text{Set of Internal States} \]

\[ \Omega = \text{State Transition Function}: \]
\[ Sb(t+1) = \Omega[Sb(t), V(t)] \]

\[ \Sigma = \text{Output Function}: \]
\[ W(t+1) = \Sigma[Sb(t), V(t)] \]

Figure 3. Subautomata of Control Automaton, continued.
central processor into an operational automaton and a control automaton. The control automaton of a microprogrammable processor is further subdivided and formalized as the composition of a microprocessor automaton and a control store automaton. Ultimately, each microprogram which exists as a component of the control store automaton may itself be represented as an individual automaton. The consequent structural abstraction introduced by modeling a computer as a system of automatons allows much simplification of inherent complexity. Such simplification will assist in the selection of appropriate sets of micro-operations. Other benefits to derive from this formalization in terms of automata theory include the application of techniques for the minimization of microprograms and verification of microprogram correctness. Both constitute important avenues for further research.

A discussion of constituents for the composition of a theory of microprogramming was presented as a means of providing an intuitive understanding of conceptual entities encompassed within the domain of microprogramming. The definition of standard terms utilized within the field is now of concern.

**Definitions**

As with most fields of specialization, the subject of microprogramming has spawned the use of several technical terms. Terminology indigenous to the literature is not always consistent, but several standardized notions have evolved. It is the purpose of this section to formally specify the definition of terms referenced in subsequent
explication. Some of the definitions will assume standardized notions while others will reflect the dialectic style of this paper.

The previous section introduced the primitive, componential concepts of microprogramming. Several formal definitions of these ideas can be found in the literature (Mercer 1957; Hussen 1970; Glushkov 1964, 1965; Flynn and MacLaren 1967; Reigel, Faber and Fisher 1972; Ito 1973; Ramamoorthy and Tsuchiya 1970; Barr, Becker, Lidinsky and Tantillo 1973). The following are representative of those formulations.

Definition 1: A **Micro-operation** is the smallest, most elemental data-processing operation that is executed during one pulse time of a given machine cycle. Operations at this level usually involve basic gate control manipulations, such as a register-to-register transfer of information, and allow similar but mutually exclusive operations (other direct gating, shifting, condition testing, complementing or binary integer addition) to occur simultaneously during the well-defined timing point (Hussen 1970, p. 34; Ramamoorthy and Tsuchiya 1970, p. 166).

Definition 2: A **Micro-predicate** is a type of micro-operation and is used to test the state of the microprocessor. A micro-predicate is usually represented by an addressable state flip-flop (actual or pseudo) containing status information and is used to reflect the results of previous micro-operations (Ito 1973, p. 6; Lawson and Smith 1971, p. 736).

Definition 3: A **Micro-order** is a set of one or more micro-operations that may be executed simultaneously (Mercer 1957, p. 158).
Definition 4: A Microinstruction is a formatted control memory word that specifies which particular micro-order is to be executed, and/or determines the next microinstruction to be executed. Physically, a microinstruction consists of bit patterns encoded in a manner that determines which control lines and logic circuits are to be activated (Ramamoorthy and Tsuchiya 1970, p. 166).

Definition 5: A Microprogram is a sequence of microinstructions used to effect a single machine code or some application-oriented function. The set of microinstructions specify appropriate sequences of micro-orders and contain micro-operations which test the status of selected logical conditions. The status information directs the flow of microinstruction execution within the microprogram (Ramamoorthy and Tsuchiya 1970, p. 166; Glushkov 1965, p. 2).

Definition 6: Microprogramming, as a conceptual entity, is a technique for the design and implementation of a digital computing system's control function. With this technique, a given set of machine operations is formally interpreted and defined through an equivalent set of micro-operation sequences. The micro-operations are executed by the propagation of control signals which are organized on a word basis and stored in a control memory unit. From an engineering point of view, a microprogrammable control unit is a programmable timing generator. As a process, microprogramming embodies the selection of suitable sequences of microinstructions for the appropriate generation of control signals and micro-operation orchestration (Mercer 1957, p. 157; Husson 1970, p. 20; Reigel, Faber and Fisher 1972, p. 705; Kampe 1960, p. 212).
Definition 7: A Microcycle is the timing cycle of the control clock which performs the fetch and execution of a microinstruction. System cycle time is traditionally associated with the minimum non-overlapping time interval between successive accesses to one main memory storage location. A separate clock usually with a faster frequency than the clock cycles used for main memory synchronization is often provided for control storage accessing. Microcycle time intervals can be made more efficient by overlapped or pipelined microinstruction execution. The length of time expended by one microcycle depends upon whether the microinstruction executed during that clock cycle is a monophase or polyphase microinstruction. A monophase microinstruction produces a single simultaneous issue of control signals which represents all necessary control line excitations during one clock pulse. A polyphase microinstruction generates control levels and signals used during two or more clock pulses and consequently requires more than a single clock pulse interval for its execution (Barr et al. 1973, Davis 1974, Redfield 1971). The consequences of these two approaches will be discussed during the explanation of design strategies.

Definition 8: We will refer to a Microprocessor in this paper as that collection of hardware which implements the functions of microinstruction fetch and execution. This includes hardware associated with the operations of microword sequencing, decoding, and control signal propagation. While this use of the term agrees with its general semantic application in most of the literature on microprogramming, it is at variance with one of the meanings often employed by a portion of the
computing community. The latter meaning is encompassed within a classificational scheme which subdivides digital systems into large centralized general purpose computers (maxi), minicomputers (mini), and microcomputers (micro). A microcomputer has been described as consisting of a microprocessor, input/output devices and memory (Holt and Lemas 1974, p. 65). Within this scope, a microprocessor refers to one or more MOS/LSI (metal-oxide semiconductor/large scale integration) chips used for the implementation of basic processing functions. Such microprocessors usually are characterized by small word lengths (4 to 8 bits), limited instruction sets and slow instruction execution times (5 to 10 microseconds). They may or may not be microprogrammed (Davis 1974, Reyling 1974, Holt and Lemas 1974). We shall refer to processors of this nature as micro-miniature processors and reserve the term "microprocessor" as a means of referring to the "inner computer" of all processors which are microprogrammable. It should be recognized that microprogramming is a technique for the implementation of control mechanisms that can be applied across the spectrum of digital systems without regard to their structural size. Maxicomputers, minicomputers, and microcomputers are all amenable to various strategies of microprogramming.

Definition 9: The storage medium for microprograms is referred to as a Control Memory or Control Store. Two properties of control memory usually account for a physical distinction between the control memory and main memory; control memory is normally realized with a faster
access time and a smaller quantity of storage capacity (Flynn and Rosin 1971, p. 727; Tucker and Flynn 1971, p. 240).

The essential feature of a control store medium is the fast access time. The characteristics of a microprogramming approach make it desirable to have a memory device for microprogram storage with an access time on the same order as the primitive combinational operations of the system. Compatibility between micromemory access time and the speed of fundamental logic units is very advantageous. Once that property has been provided, it is an economic decision as to whether or not the main memory is realized through the same medium. Although an architectural merger of the two memory units is possible (Haynes 1972, p. 61), a conceptual distinction is useful and shall be maintained for the purposes of this paper. Any medium that provides residence for microprograms during their interpretation and execution is by definition the control store.

Closely related to the design decisions regarding control memory size and access time is the problem of micromemory changeability. This is largely dependent upon the memory technology employed. The types of memory technologies which are adaptable as media for the control store function are classified into two categories: read-only memory (ROM) and writeable control store (WCS). A read-only memory has been characterized (Hill and Peterson 1973, p. 69) as a device in which the stored information (microprogram) is either permanently fixed during fabrication or can be altered only by mechanical changes to the device structure. A writable control store, however, allows electronic alteration of the
microprogram stored in control memory. Read-write micromemory capabil-
ties can be provided by conventional random access memory (RAM) elements
or by units which possess fast read, slow write characteristics. De-
pending upon the particular memory technology selected, a microprogram-
mable system is classified as either a static or dynamic system.

Definition 10: A static microprogramming system implements the
control function by means of microprograms permanently fixed within a
read-only memory (ROM) control storage medium. Once the ROM circuit
board is connected to the system, the resident microprogram dictates
the available instruction repertoire and the machine structure becomes
static (Cook and Flynn 1970, p. 213).

Definition 11: A dynamic microprogramming system possesses a
read-write control storage medium which allows automatic, electronic
alteration of resident microprograms. This capability enables the system
to implement more expansive microcode routines and many machine vo-
cabularies (Cook and Flynn 1970, p. 213). This classification essen-
tially measures a degree of hardness realized in the design. A static
microprocessor possesses a greater degree of hardness than one which
is dynamic. Under current technologies, it will usually be the faster
of the two.

Several terms native to the broad discipline of computer science
are also relevant to the specialized view of microprogramming. The
concept of emulation provided one of the original architectural bridges
to the philosophy of microprogramming. Initially, the emulation function
was closely related to the process of simulation. Simulation procedures
were employed as a means of approximating the functional behavior of one computer by another computer. Each machine language instruction of the virtual or target computer would have to be expressed in terms of an equivalent sequence of machine instructions capable of being executed by the host computer. Because of the nature of fixed machine language instruction sets for general-purpose computers, this simulation process was not very efficient. Unless the instruction formats and repertoire of the two machines were similar, development of mapping functions for duplicating the target machine's behavioral responses to inputs on the host machine could be extremely agonizing.

One of the initial popular areas of application for microprogramming developed in this situation. The architectural features of microprogramming allowed greater efficiency in the development and execution of machine simulating procedures. The reconfiguration of a computer through the use of microprograms so as to allow it to execute machine language programs written for other computers became known as emulation.

Definition 12: Emulation is the ability to execute machine language programs intended for one machine (the emulated machine) on another machine (the host machine) (Reigel, Faber and Fisher 1972, p. 715).

A perspective which encompasses more generality than the above definition has recently been suggested. Frieder (1973) postulates that machines should be regarded as concepts which can be realized in many ways. A particular hardware realization of a machine, which is in fact
an emulation of itself, represents only one of many possibilities (Frieder 1973, p. 66). Accepting this viewpoint, the entity which accomplishes the emulation is characterized as a procedural process, and the following definition is offered.

Definition 13: An **Emulator** is a process which is applied to a machine input set to produce an output set (Frieder 1973, p. 66). The terms "input set" and "output set" refer to the set of symbols and/or signals used in the transfer of information in the form of data and programs.

The composition of microprocessor hardware and the set of microprograms used to define a target machine language is often referred to as an emulator. This usage introduces a term borrowed from the traditional dialect of software: the concept of a **macro**. The development of assembly languages and assemblers included a facility for "macro-processing." This feature allowed a programmer to refer to a group of instructions as though they were a single instruction (Waite 1967, p. 433). When the assembler encountered the name of a macro-instruction during the assembly of a symbolic machine language program, all component instructions of the macro would be inserted at that point of the in-line object code.

The development of macro-processors provided increased powers of expression to programmers. By specifying appropriate sets of macro-instructions, they could effectively tailor an assembly language into higher level programming structures (Waite 1967, p. 433). A functional collection of microinstructions used to define a particular
machine operation can be characterized as a macro. In this sense, the term macro is similar to a microprogram. The following definition is provided for clarification.

Definition 14: A Macro or Macro-operation is a type of microprogram which is used to define a machine language operation. It consists of a functional set of microinstructions and is treated as a re-entrant subroutine. It may be called by any instruction within an application program (in main memory). After execution of the macro, control is returned back to the next instruction of the application program. Microprogrammed macros can be nested in that any active macro may call any other macro, but they are not usually recursive (Tucker and Flynn 1971, p. 246).

Another term which possesses several levels of meaning, when used within the context of microprogramming, is the notion of interpretation. The term's usual sense of usage within the broad sphere of computer science refers to the translation of a high-level language program into an intermediate or internal form. The execution of the program proceeds directly from this intermediate form without further translation into a machine language (Reigel, Faber and Fisher 1972, p. 720; Lawson 1968; p. 477). The interpretation process may proceed instruction-by-instruction, translating and executing one instruction before evaluating the next one in sequence, or it may translate all instructions into intermediate form before executing them.

Within the context of controls, interpretation denotes a process of evaluating individual instruction formats and initiating command
signals that will effect the functional definition of the instruction. Since microprogramming encompasses both spectrums of meaning, the following definition is proposed for its generality.

Definition 15: Interpretation is a process which analyzes the form of an input item and effects the prescribed meaning of the form by specifying the execution of available functional operations. An Interpreter is a procedure or entity which accomplishes the above process.

The above discussion of form and meaning envelopes terminology used in the description of linguistic expressions. It is customary to refer to the structural form of an input string or sentence of some language as its syntax (Caracciolo 1965, p.224; Feldman and Gries 1968, p. 104). The meaning that is assigned to the string's syntactic structure is known as its semantics. When concerned with the formal definition of a programming language, the notion of semantics involves the specification of the manner in which well-formed (with respect to its syntax) statements of the language are executed (Feldman and Gries 1968, p. 79). In this sense, semantics concerns the effect that some syntactic structure has an input to a real or abstract machine. The set of microprograms used to define a machine language could be regarded as the semantics of a computer. This perspective and other aspects will be discussed in further detail during a subsequent section on language processors.

Finally, it is necessary to formally clarify the three different implementation levels for the realization of information processing functions on digital computers. The differentiation between hardware
and software was quite distinct in the earlier stages of computer development. The advent of microprogramming has helped to blur that distinction. As the aspects of various programming language features and data processing functions become better understood, their means of realization often shift from software to either firmware or hardware.

Definition 16: **Hardware** refers to hardwired circuits, memory elements, data paths, switching facilities and logic units used to mechanize the (fundamental) computational abilities of a digital computer. The set of functional units and their control implemented as hardware represent the physical resources available for computational processes and as such prescribe the physical boundaries (processing speed, storage capacity, computational power, etc.) to their domain. Since these resources are implemented as hardware, they are not modifiable (with any ease or economy) after fabrication. The complexity of the basic set of operations realized in hardware is a design variable. Language and architectural functions such as stacks, procedure calls and various addressing schemes have been and are suitable to hardware implementation.

Definition 17: The mechanization of information processing algorithms as programs expressed in terms of some programming language results in **software**. The spectrum of programming languages ranges from the machine language level to numerous species of high-level languages. Although all programs generated from such programming languages are referred to as software, there are two general classifications of program categories: application software and system software. **Application software** includes programs which are developed for some problem solving.
task endemic to the physical world, or environment outside the digital processing system. System software consists of programs used to manage the physical resources of hardware, creating an operating environment within which application programs can be executed.

Definition 18: Firmware designates the set of microprograms which may reside in control store (Opler 1967, p. 22). Firmware as such can include microprograms used either for control and machine structure definition or for application procedures. In other words, one could construct both system and application firmware.

This concludes the definition of terminology relevant to the discussions of this paper. An introduction to the concepts of microprogramming both from a historical and theoretical perspective was presented in Chapter 1. In Chapter 2 we begin to look at the actual mechanization of these ideas.
CHAPTER 2

THE ORGANIZATION AND DESIGN OF A MICROPROGRAMMABLE CONTROL ENGINE

The previous chapter identified the conceptual constituents which comprise the topic of microprogramming. The present chapter is directed towards the development of an explanatory microprocessor model in order to illustrate the hardware resources necessary for the construction of a microprogrammable processor. Using the microprocessor model as a pedagogic tool, an explanation of design strategies involved in the structure of possible microprocessor architectures follows.

A Microprocessor Model

Machine architecture and organization is structured around particular combinations of hardware: registers, buses, transformational networks and controls. The functional domain of a machine, whether microprogrammed or not, is specified by the set of computational operations and data manipulations it is to accomplish. A computing machine becomes completely defined then when its register structure, its information flow schema and its operations are formally specified (Proctor 1964, p. 423). The conceptual distinction between a computer’s operating functions and its control functions, as developed in the automata model of Chapter 1, will be maintained during the discussion of the physical realization of these functions. The operating functions are embodied
in an operating engine, while the control functions are embodied in a control engine.

An operating engine usually contains a group of general-purpose registers, and various functional units such as adders, shifters and combinational logic for boolean operations. Data paths between the registers and functional units are constructed either by means of separate AND and OR gates or through the use of buses (Hill and Peterson 1973, pp. 77-90). Registers and single state flip-flops within the operating engine are often referred to as local storage (Kleir and Ramamoorthy 1971, p. 784). Machine storage devices of this category are analogous to machine program variables. Operating engine functional units perform data manipulations upon values in the registers and return the results, as well as side effects, to specified destination registers. A random access memory unit provides the main memory capacity for the operating engine.

The architectural structure and functional characterization of the operating engine can be considered independent from the design philosophies employed in the construction of the control engine. The number of internal registers and their interconnections, the type of elementary arithmetic and logical operations performed, and the incorporation of shifting paths, gates and buses are determined primarily by the functional objectives of the system. The manner in which the control engine is implemented does not directly influence the physical realization of the operating engine (Vandling and Waldecker 1969, p. 45). However, the structural and functional characteristics of the operating
engine exert considerable consequences on the many design decisions encountered in the construction of the control engine. The operational capacities of the arithmetic and logic units, the access time of main memory, and the data routing paths provided, all apply certain constraints upon the set of micro-operations that can be defined, and the manner in which their execution is controlled (Kleir and Ramamoorthy 1971, p. 784).

In a microprogrammable processor, the control engine generates sequences of control signals (voltage pulses or voltage levels) by selecting and decoding microinstructions from the control memory. State information for the microprocessor is maintained by microprograms stored in the control memory and by means of addressable state flip-flops. State information about the processor being emulated is derived from pseudo state flip-flops of the operating engine. This information is usually represented in the bits or groups of bits of local storage registers such as the instruction register, the accumulator or several index registers.

The consequences of adhering to a microprogramming design philosophy for the implementation of a digital system control function are reflected in the structure and organization of the control engine. An example of the internal composition of control and operating engines is illustrated in Figure 4. The control engine represents a microprogrammable control function (automaton). A non-microprogrammable processor would merely embody the control function in a large, complex sequential circuit similar to that depicted in Figure 5.
Figure 4. Microprogrammable Processor Model.

Figure 5. Conventional Processing Unit.
The operating engine is modeled after the Small Instructional Computer (SIC) described in Hill and Peterson (1973). The SIC architecture represents a fairly primitive machine organization, but its simplicity was selected for pedagogic and compatibility purposes. The SIC hardware components consists of an instruction register (IR), an accumulator (AC), a memory address register (MA), a memory data register (MD), a program counter register (PC), two index registers (IA and IB) and a status/link flip-flop (L). The utilization of buses for the implementation of data paths provides greater flexibility (at some cost in execution speed) and it is this manner of register and functional unit interconnection that is selected for the model operating engine in Figure 4. A more precise illustration of operating engine structure at the register and bus level is depicted in Figure 6. The arithmetic logic unit (ALU) of Figure 4 has been subdivided into an adder, a shifter, an incremener (INC) which adds one to the operand, and logical units for the AND and EXCLUSIVE OR (XOR) functions. Three buses, the ABUS, BBUS and OBUS, realize the data path connections.

Control engine components include a microsequencer, a control memory unit, a microinstruction register (MIR), a micromemory address register (MAR) and a microinstruction decoder. The control memory could be either ROM or RAM. The microprocessor (automaton B) of Figure 3 has been separated into the microsequencer, decoder, MAR and MIR parts of Figure 4. The complexity of the microsequencing and decoding units will vary with the particular microword design philosophies applied to the specification of microinstruction formats. The microsequencer's function
Figure 6. Register--Bus Data Paths of Model CPU Configuration.
is to direct the transfer of information vectors (microword bit patterns) between the control memory, MIR, MAR and the decoder. The principal aspect of this operation becomes the determination of the chronological order in which microinstructions are to be executed. Using status information from the pseudo flip-flops of the operating engine and microprogram state information reflected by bit fields of the microinstruction currently in MIR (in addition to other possible microprocessor addressable state flip-flops), the microsequencer determines the proper sequencing for microprogram execution.

There have been two traditional approaches to the means of sequencing microinstructions. One scheme known as current word sequencing (Barr et al. 1973) uses the technique whereby each microinstruction determines the address of the next microinstruction to be executed. The other procedure specializes this functional aspect of microinstructions by specifying two (or more) classes of microword formats. One format is defined for the purpose of stipulating register transfer information while a separate format containing branching and sequencing information is used. This results in at least two distinct types of microinstructions, one to control register transfers and logical manipulations of information stored in the operating engine registers, and another to control the sequencing of microinstruction execution and manipulation of registers and state flip-flops of the microprocessor (control engine). Consequences of these two approaches are discussed in the next section on design strategies.
Essentially, the microsequencer can be structured as a small hard-wired control sequencer similar to the complete hard-wired version of Figure 5 (Hill and Peterson 1973, pp. 230-253). In an organization similar to that of Figure 4, the microaddress register serves as both a microprogram counter and control memory address register. By providing additional registers in the microsequencer design, the more complex effective address calculation capabilities such as control memory indexing and stack processing could be included in the microprocessor architecture. A microprocessor possessing such capabilities would indeed resemble a computer within a computer, but the increased complexity, although allowing greater flexibility, would of course produce an increase in implementation cost. Chapter 3 will entail a further explanation of microprocessor design and architecture as it is related to possible microprogramming languages and the computational abilities and processing objectives assigned to a system's operating engine. At this point we have been primarily concerned with a simplified model which would assist in a general understanding of microprogrammable computers.

Design Strategies

The organizational aspects of a microprogrammable control engine will reflect the functional constraints of the operating engine. But within this structural framework there remain a number of significant design tradeoffs which must be considered. The definition and specification of a control memory structure represents the most influential source of design choices for the construction of a microprocessor (Dollhoff 1973, p. 95). The manner in which solutions to control memory design problems
are selected will determine the ultimate form of the microprocessor and
general characterization of the control engine.

Design specification for control memory involves the determination of its bit and word dimensions. A control memory's bit dimension represents the length or number of bits available for the microinstruction format, while its word dimension reflects the number of microinstructions that can reside within control memory. Both of these measurements determine control memory's physical size and consequently have a direct bearing on the unit's cost. The bit dimension is perhaps the most critical aspect however (Hoff 1972, p. 63). Although there have developed two distinct philosophical approaches towards the structuring of control memory dimensions and microinstruction formats, each approach involves the conjunction of strategies for several distinct but interdependent objectives. This section proceeds with a detailed explication of objective design goals and the conceptual mechanism available for their realization.

The operating engines of most information processing systems are structured in a hierarchical organization of functionally primitive resource partitions. A system's data processing is performed by the coordination of functionally distinct, elementary units such as adders, shifters, registers, main memory, and the addressing mechanisms of storage resources. The execution of a machine code operation involves the transfer of information between and among these functional components. This transfer of information is accomplished by the opening and closing of gates and circuits between registers and basic logic elements
(Ramamoorthy and Tsuchiya 1970, p. 166). Operations on the gates and circuits at this level represent the primitive machine functions or micro-orders of the operating engine.

The development of a microinstruction format which provides for the efficient manipulation and management of these primitive resources becomes the broad objective of microword design. It is desirable that the format selected allow for the control of internal data flow to each one of the partitioned resources, as well as providing for communication between them, during a single microinstruction execution (Flynn and Rosin 1971, p. 728).

Considerations of this nature produce several problems for the design of a microword format. Vandling and Waldecker (1969, p. 46) have identified the following as fundamental design questions:

1. How many different control functions are to be specified by a single control word?
2. How are different control functions to be interrelated; are they to be performed sequentially, concurrently, or in some combination?
3. How is the address of the next control word derived?
4. How are control words to be encoded?

Answers to these questions ultimately determine the form of control word format and microprocessor structure. The number of distinct control functions and their interrelation are specifically related to characteristics of the operating engine. Organizational aspects of the operating engine (which are determined by functional goals of the system) consequently influence the domain of microword design. But the microword of any microprogrammable control engine will have to perform
a number of general operations. P. M. Davies has provided an informative discussion on the basic functions of microinstructions and the following discourse parallels his presentation (Davies 1972, pp. 23-24).

Each microword (or sequence of microwords) must incorporate information requisite to the generation of level or pulse signals on appropriate control lines for each time cycle. The representation of information necessary for the establishment of control line settings becomes a principal design decision in the selection of a suitable format. Another important microword function is the ability to control the manipulation and evaluation of addressable state flip-flops and condition recording registers. This ability to store and test state information is useful in a third microword function; that of establishing the address of the next microinstruction to be executed. Other functions (such as control memory fetch and store operations for a WCS, or a constant emit field for the introduction of numeric values into the data flow from control store) can be specified in the design of a microword, but the three basic operations involve the establishment of control line signals, the testing and setting of state information, and the generation of the next microinstruction address (Davies 1972, pp. 23-24). We will now consider the means for the realization of these functions within the structure of control memory and microprocessor organization.

Two distinct design philosophies have emerged as strategies for the construction and utilization of microinstruction formats. The two microprogramming techniques have become known as the horizontal (hard-view) and vertical (softview) approaches. Although the two perspectives
are essentially concerned with the number of resources which each microinstruction controls, distinctions between the two philosophies are further characterized by three measures of reference. The first measure is concerned with the length of time (number of clock pulses) for which an individual microinstruction remains active and valid. This measure divides microwords into the monophase or polyphase type of microinstruction. A monophase microinstruction generates control signals used during only one clock pulse, whereas a polyphase microinstruction remains valid for two or more clock pulses and could perhaps continue to generate control levels or signals used during several clock cycles (Redfield 1971, p. 743; Dollhoff 1973, p. 95; Agrawala and Rauscher 1974, p. 820).

From the perspective of a microprogram, a monophase microinstruction would result in a single, simultaneous issue of control signals while a polyphase microinstruction issues a series of control signals over a span of time, subdividing each major clock cycle into multiple minor clock cycles.

If the number of micro-operations that can be executed simultaneously during one clock pulse is relatively small, then a rather large number of monophase microinstructions would be required in order to implement a macro-operation. This implies a large number of control memory accesses which increases the amount of execution time (Redfield 1971, p. 743). Although the purpose of the polyphase approach is to decrease processor execution time, it also increases both microword width and control memory addressing logic (Dollhoff 1973, p. 96). Microword format complexity and execution time increase as the valid active time span of
a polyphase microinstruction lengthens. If the same control signal must be generated at different times by one polyphase microinstruction, several additional bits must be used in order to distinguish between the different time requirements. This represents an inefficient utilization of the microword's information capacity and results in a larger control memory (Redfield 1971, p. 744). Since a polyphase microinstruction becomes specialized to particular instances of micro-operations, there are fewer microinstructions that can be shared by more than one macro-operation. Hence, a larger repertoire of microinstructions will be required. In view of these constraints, the maximum efficient period of time for a polyphase microinstruction to remain valid is usually one system clock cycle (one complete set of clock pulses) (Redfield 1971, p. 744).

Thus while a monophase technique normally requires a larger number of microinstructions per microprogram, it allows for a shorter microword and a smaller set of microinstructions for a microprocessor's instruction repertoire. When constraints on the cost, size and simplicity of the control memory exceeds the need for execution speed, this approach is more appropriate. Polyphase microinstructions result in fewer microwords per microprogram and faster execution times at the cost of larger control memory size and increased complexity. When design objectives can absorb greater degrees of complexity and cost in the quest for faster execution time, the polyphase strategy or some combination of polyphase and monophase can be selected.
A second measure of reference that determines the philosophical direction of microword construction refers to the degree of encoding in the microinstruction format. This design measure provides the greatest influence on the size or width of the microword. Encoding involves the manner in which the control functions are represented in the microinstruction. A particular representational scheme will associate some set of bit fields to directly control the functional hardware facilities of the operating engine. Strategies for the assignment of bits to functional fields are bounded by two approaches and can be measured on a continuous scale. There is an inverse relationship between the microword's bit width and the amount of encoding/decoding required for execution. Schemes that employ the least encoding require the maximum number of bits, whereas field assignments with much encoding need fewer bits (Redfield 1971, p. 745; Hoff 1972, p. 61; Vandling and Waldecker 1969, p. 51).

At one extreme there is a one-for-one association between each bit of the microword and its assigned control line. Since each bit represents an individual control function, no encoding is utilized. A bit assignment of this nature is not feasible for any but the simplest of machine architectures. As the number of required control signals increases, the width of the microinstruction must increase and this results in a higher cost for control memory. The number of bits in control memory can be decreased by identifying groups of control signals which are logically mutually exclusive (Dollhoff 1973, p. 95; Davies 1972, p. 23).
The fact that only a small fraction of the possible combinations of control signals represent meaningful functions makes encoding possible. A group of commands can be identified as logically mutually exclusive if it is the case that only one control line from the group is activated at any one time during meaningful control states. Each such set of control signals can be encoded and assigned a bit field in the microword. Commands to a processor are either explicitly or implicitly mutually exclusive.

A set of control signals would be explicitly mutually exclusive in the sense that if one of the command signals was propagated, the other commands in the set should not be or could not be generated. A set of commands to gate one of several registers out to a common bus or a set of commands to select one of several inputs to a common register would be examples of such mutually exclusive command sets (Hoff 1972, p. 61). The grouping of these commands into mutually exclusive sets places no restrictions on the processing flexibility of the operating engine that does not already exist in the hardware. A set of implicitly mutually exclusive control signals could be constructed from commands in which there were no existing requirements for their concurrent execution. Construction of such implicitly mutually exclusive command sets should proceed carefully so that control signal combinations which may be of value in future microprogramming applications are not prevented or excluded by the encoding scheme (Hoff 1972, p. 61).

The other extreme of control line/microword bit assignment is represented in the extensive application of encoding. Under such a
design discipline every bit is encoded to the fullest extent possible without the exclusion of essential control signal combinations. This provides for the maximum utilization of a microinstruction's information capacity. While a substantial amount of encoding will produce a short microinstruction and thereby permit a smaller control memory storage medium, this approach has a number of possible disadvantages associated with it. The decoding functions which accompany highly encoded microword formats will necessitate the inclusion of additional hardware for decoding circuitry. The decoding process itself will expend a certain amount of time and consequently produce slower microinstruction executions (Dollhoff 1973, p. 95; Redfield 1971, p. 745). For some system environments, desired combinations of particular control signals may not be mutually exclusive. A high degree of encoding under such circumstances will usually restrict the manner in which various micro-operations can be combined. This represents a loss in microprogramming flexibility in that certain nonessential but nice to have combinations of control signals and micro-operations become impossible (Redfield 1971, p. 745).

The specification of a microinstruction's bit dimension then is bounded by the two approaches to field assignment. A one-to-one allocation which assigns one bit to control each micro-operation represents the upper bound and as such will require the maximum number of bits and the minimum amount of field decoding. The lower bound on microword bit width is achieved by defining a single field of n bits which encodes the operation for $2^n$ possible clock pulses (Vandling and Waldecker 1969, p.45). This type of format requires the minimum number of bits and the maximum amount of decoding.
A decision as to what particular degree of encoding represents an optimum format specification depends on several criteria. The principal influence is whether or not the microprocessor is being designed for horizontal (hard) or vertical (soft) microprogramming. Horizontal microprogramming and the small amount of encoding which it implies provides a greater opportunity for the parallel and simultaneous execution of micro-operations. The characteristics of concurrent micro-operation specification and short execution time periods expended in the decoding process make the direct control (little encoding) scheme attractive as a microprocessor design strategy for control engines which require an optimum generation and orchestration of control signals. The formulation of microcode for a horizontal microprocessor however will be extremely detailed and somewhat tedious. Vertical microprogramming on the other hand is more concerned with the design of a microprocessor which can be easily microprogrammed. Such a philosophy usually concentrates on the definition of a somewhat universal set of micro-operation combinations which can be realized as microinstructions. As a consequence microinstruction sets for vertical microprocessors are constructed at a slightly higher level than those of horizontal machines and are applied algorithmically as in conventional programming (Dollhoff 1973, p. 96).

The third reference measure useful in the taxonomy of microinstruction and microprocessor design philosophies has to do with the manner in which control memory address generation and sequencing is determined. This functional task of the control engine is associated with methods of representation for sequencing information within a microword, and with
the time relationship between execution of a current microinstruction and address generation and selection of the next microinstruction. The availability of conditional branch tests and specification of microinstruction addresses are features which profoundly influence system cost and performance (Barr et al. 1973, Hill and Peterson 1973, Husson 1970). The over-all machine speed of a computer which has a microprogrammable control engine will be limited by the access time of its control memory. Procedural methods of control store address generation and accessing will determine the size of the control memory and the number of times it has to be accessed during execution of any one microprogram. The number of conditional tests upon which branching is based provides a measure of flexibility embodied within the structure of the microprocessor.

Actual control word addresses may be determined by information from function codes, status flip-flops, and counter settings which represent the general state of the operating engine's data path logic as well as by information on the control engine's state brought about during previous microinstructions. A microword's width will be directly dependent upon whether or not a single microinstruction is used to contain information necessary for both register transfer/data manipulation operations and branching operations. Microprocessors which employ "current word sequencing" utilize a microinstruction format that includes sufficient information for both branch and transfer operations in one microword. This decreases the number of times control memory is accessed during execution of a microprogram.
Whether or not branch and transfer operations should be combined in one microinstruction is dependent upon the number of branch conditions that can be set and tested, the frequency of branch operations expected in microprograms to be executed by the microprocessor, and the cost of lengthening the number of bits per microword (the control store width). Hill and Peterson (1973) report that microprograms will usually include a large number of branch operations, although this characteristic is probably closely related to the specific nature of a microprogram's application. Microprograms in which the frequency rate of branching approaches 50 percent (every other microinstruction) would be more efficiently executed if each microinstruction specified the next microinstruction.

A distinctive feature which characterizes the structural flexibility of a microprocessor is the size and variability of branch conditions that can be selected and tested. Hill and Peterson (1973, p. 242) believe that the difficulty of providing sufficient branching capability is a basic limitation of microprogramming. They point out that a hardwired control unit implementation makes it possible "to connect directly to any bit or combination of bits, of any register in the machine" in order to specify a branch condition (Hill and Peterson 1973, p. 254). Of course any particular hardwired control sequencer selects only a certain subset from this range of possibilities and is forever thereafter limited to that mode of operation. The philosophy of microprogramming however dictates that the binding time of such architectural relationships be delayed for specification by the microprogrammer. It is therefore
useful to provide the microprocessor with as many branching conditions as possible. This of course will result in wider microinstructions as more bits are required to represent and direct the status and branch information. Depending upon size constraints for control memory, it may be more feasible to separate branch operations from register transfers and data manipulations by defining two or more types of microinstructions to control each. Such a strategy will add to microprogram execution time since additional accesses to control memory will be encountered.

Microprogram execution times can be decreased by overlapping the next microinstruction's fetch with the current microinstruction's execution. This mode of operation has been referred to as the parallel approach (Redfield 1971). Control store addresses may be calculated from several sources including the current microinstruction address field, an address counter, status flip-flops and function codes. The overlapped fetch and execution of microinstructions requires that branch decisions be made on information available at the end of the data manipulation performed by the microinstruction preceding the current microinstruction. It may therefore be necessary to ignore a microinstruction that was fetched prior to the determination of the branching condition, and to generate another address and microinstruction fetch. This results in a slower execution time, especially if the microprogram possesses a large number of branch points (Redfield 1971, p. 744).

The serial philosophy of microinstruction execution results in an appropriate control memory access every time since generation of the next
microinstruction's address is always done after execution of the current microinstruction. However, in a serial approach, the data manipulation specified by a microinstruction is delayed by the amount of time it takes to access the control store. Each time a new microinstruction is needed during the course of a macro-operation's execution, the total execution time is increased by an amount equal to the effective access time of control memory. The accumulation of control memory accessing time may be able to be absorbed without affecting the efficiency of the macro-operation if the access time is relatively small when compared with operation times of the logic function manipulators, or if the macro-operation itself is subject to some external constraint, such as main storage access time. When considering a microprogrammed implementation of a machine instruction set, Redfield (1971, p. 744) proposed the following means of choosing between the parallel and serial approaches:

If the data path time plus the ROM effective access time add up to less than the main storage access time, then the serial philosophy seems very practical for this application provided main storage is used in most instructions.

The parallel approach, if the system execution is not limited by external constraints, appears to be better than the serial approach providing the expected execution time loss due to microprogram branching is less than the sum of the effective access times for the required microinstruction.

It is the confluence of design decisions among the strategies of direct versus encoded control, monophase versus polyphase, serial versus parallel, and branching specifications which emerges from and is shaped by the larger philosophical design perspective towards microprocessor structure. A horizontal or hard viewpoint to microprogramming will seek to design a system which fully exploits the parallel-processable
opportunities available. Its objectives are directed towards the development of a fineness of control which allows a compact and efficient specification of macro-operations. As such, it strives to realize the most efficient manipulations of data path logic in order to minimize the macro-operation execution time. A philosophy of this type normally results in wide, polyphased, serially executed microinstructions with direct control or little encoding. The branching information is also usually specified by each microinstruction in the same microword. A microprogram which is written with horizontal microinstructions will be characterized by a small number of wider words when compared to a similar program composed of vertical microinstructions (Redfield 1971, Ramamoorthy and Tsuchiya 1970, Dollhoff 1973).

A vertical or soft viewpoint to microprogramming involves a different emphasis on what class of microprocessor features should be optimized. This philosophy seeks to construct microprocessors with small control memories and simple control addressing logic. Rather than concentrating on highly efficient macro-operations, vertical microprogramming is concerned about optimum algorithmic structures for microcode; this implies a relatively small and simple microinstruction set that can be used very efficiently. Objectives such as these are realized with monophased, highly encoded microinstruction sets which often include separate branch-type instructions. In order to offset the longer execution time which results from increased decoding and control storage accessing, vertical microinstructions are frequently executed in parallel (Redfield 1971, Ramamoorthy and Tsuchiya 1970, Dollhoff 1973).
HORIZONTAL FORMAT

A field size of 1 bit represents no encoding

<table>
<thead>
<tr>
<th>Type</th>
<th>Tag</th>
<th>OPCODE</th>
<th>operand 1</th>
<th>operand 2</th>
<th>operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>source register</td>
<td>source register</td>
<td>result register</td>
</tr>
</tbody>
</table>

REGISTER TRANSFER INSTRUCTION

<table>
<thead>
<tr>
<th>Type</th>
<th>Tag</th>
<th>BRANCH Function or Condition Test</th>
<th>CONTROL MEMORY ADDRESS</th>
</tr>
</thead>
</table>

BRANCH INSTRUCTION

<table>
<thead>
<tr>
<th>Type</th>
<th>Tag</th>
<th>REGISTER</th>
<th>FUNCTION</th>
<th>CONSTANT</th>
</tr>
</thead>
</table>

LITERAL INSTRUCTION

VERTICAL FORMAT

Figure 7. Microword Formats.
contains an example of generalized microinstruction formats which exemplify the two philosophical design strategies, while Figures 8 and 9 illustrate possible vertical and horizontal microinstruction formats for the previously defined microprocessor model.

A conceptual and structural merger of horizontal and vertical microprogramming was proposed by Rosin, Frieder and Eckhouse (1972) with their introduction of the term "nanoprogramming." It was their opinion that the two design philosophies could be usefully combined through the concept of "two-level emulation" (interpretation). They would add further flexibility to the architecture of a microprocessor with the inclusion of a second level control memory categorized as "nanostore." This nanomemory would be constructed as a medium for long, horizontal nanoinstructions which specified primitive micro-operations of gate-level controls. The micromemory on the other hand would embody short, vertical microinstructions which essentially are characterized as addresses into the nanostore. They describe a hierarchical structure in which a virtual machine instruction is interpreted as a sequence of microinstructions which are individually in turn interpreted by the execution of a nanoinstruction group. However, the microprogram in this case exists only as subroutine calls or pointers into nanostore.

The definitive objective of highly parallel and efficient groups of micro-operations is attained through suitable specification of a horizontal nanoword, while the goal of a small algorithmically efficient set of vertical microinstructions is realized by means of an appropriate collection of nanostore addresses. It was envisioned that both
Figure 8. Horizontal Microinstruction Format for Model Microprocessor.
Figure 9. Vertical Microinstruction Format for Model Microprocessor.
microstore and nanostore would be writable thus allowing for the redefini-
tion of the microinstruction set (Rosin, Frieder and Eckhouse 1972,
p. 750). The merits of this approach have yet to be determined. A
higher degree of flexibility is obtained by the introduction of another
level of interpretation, but the associated disadvantageous (slower
execution and a deeper level of complexity in program specification) are
also incurred. An environment which finds such amounts of flexibility
desirable may be willing to accept the cost. Research projects on the
structure of specialized machine architectures would be an example.

This concludes the discussion about microprocessor resources and
organizational structure. The means of implementing the three principal
microinstruction functions (propagation of control line signals, testing
and setting of state information, and address generation of the next
microinstruction) depend upon the characterization of the microprocessor
as either horizontal or vertical. This classification is essentially
determined by decisions which reflect the structure and interpretation
of a microinstruction's format. As described, the number of machine
cycles used to generate control pulses from one microinstruction (mono-
phase or polyphase), the amount of encoding in the association of bit
patterns to control functions, and the techniques of microinstruction
sequencing (overlapped fetching and branch-type microinstructions) all
possess specific implications for the characteristics of a microprocessor.
Tables 1 and 2 summarize these implications.
Table 1. Design Strategies and Implications for Microinstruction Formats.

<table>
<thead>
<tr>
<th>Format</th>
<th>More Instructions per Microprogram</th>
<th>Longer Execution Times</th>
<th>Shorter Microwords</th>
<th>Smaller Set of Microinstructions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MONOPHASE</strong></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td><strong>POLYPHASE</strong></td>
<td>* shorter microprograms</td>
<td>* shorter execution times</td>
<td>* longer microwords</td>
<td>* larger microinstruction sets</td>
</tr>
<tr>
<td><strong>DIRECT CONTROL</strong></td>
<td>* more concurrent execution of micro-operations</td>
<td>* faster execution—no decoding</td>
<td>* longer microinstructions</td>
<td>* difficult specification of microprograms</td>
</tr>
<tr>
<td><strong>ENCODED</strong></td>
<td>* less parallelism</td>
<td>* longer execution times</td>
<td>* shorter microinstructions</td>
<td>* easier to select correct sequences for microprograms</td>
</tr>
<tr>
<td><strong>CURRENT WORD SEQUENCING</strong></td>
<td>* longer microinstructions</td>
<td>* shorter microprograms</td>
<td>* fewer accesses into control store</td>
<td>* fewer possible branching conditions</td>
</tr>
<tr>
<td><strong>SPECIALIZED BRANCH</strong></td>
<td>* shorter microinstructions</td>
<td>* longer microprograms</td>
<td>* more accesses into control store</td>
<td>* larger set of branching conditions</td>
</tr>
</tbody>
</table>
Table 2. Characteristics of Horizontal and Vertical Microprocessors.

<table>
<thead>
<tr>
<th>HORIZONTAL MICROPROCESSORS:</th>
</tr>
</thead>
<tbody>
<tr>
<td>* wide microinstructions</td>
</tr>
<tr>
<td>* shorter microprograms</td>
</tr>
<tr>
<td>* little or no encoding</td>
</tr>
<tr>
<td>* polyphased, serial execution</td>
</tr>
<tr>
<td>* current word sequencing</td>
</tr>
<tr>
<td>* fast execution times</td>
</tr>
<tr>
<td>* larger sets of microinstructions</td>
</tr>
<tr>
<td>* more expensive control memory</td>
</tr>
<tr>
<td>* hard to microprogram</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VERTICAL MICROPROCESSORS:</th>
</tr>
</thead>
<tbody>
<tr>
<td>* short microinstructions</td>
</tr>
<tr>
<td>* longer microprograms</td>
</tr>
<tr>
<td>* highly encoded</td>
</tr>
<tr>
<td>* monophase, parallel execution</td>
</tr>
<tr>
<td>* branch type of microinstructions</td>
</tr>
<tr>
<td>* slower execution times</td>
</tr>
<tr>
<td>* a small standard set of microinstructions</td>
</tr>
<tr>
<td>* less expensive control memory</td>
</tr>
<tr>
<td>* easier to microprogram</td>
</tr>
</tbody>
</table>
CHAPTER 3

ON THE CONCEPTUALIZATION OF MACHINE STRUCTURE

The interrelationships among the hierarchical concepts of microprogrammable processors, microprogramming languages, and microprogramming applications will be the subject of consideration in this chapter. A principal conjecture to be made is that microprocessor architecture is primarily determined by the nature of the applications to which the micro-code will be applied and that these aspects will in turn determine the characteristics of the software or programming language facilities which can be made available for the writing of microprograms. Computational tasks which have been identified as appropriate for microprogrammed implementation and the manner in which microprograms (firmware) can be integrated within the hierarchies of system structure are discussed. Some considerations as to the implications and influences of various microprogrammed algorithms upon satisfactory designs of micro-assemblers and higher-level microprogramming languages are also presented.

The Hardware-Firmware-Software Mix and Microprogramming Applications

The elements of machine structure (main memory, control storage, local/register storage, functional units and data paths) constitute the basis from which hardware and software designers attempt to construct
appropriate and useful virtual systems. Virtual system structure is essentially composed of four interrelated but often conflicting subcomponents: (1) machine hardware, (2) operating systems, (3) language processors and (4) user programs. These may exist as a hierarchy of routines in hardware, firmware, and software. From the broad perspective, hardware and software designers are algorithm designers; it is only the physical realization of the algorithm which differs (Flynn and Rosin 1971).

Hardware is the logical primitive in this hierarchy since any computable function may be embedded or implemented by a sufficiently complex network of logic gates and circuitry. Prior to the availability of microprogramming, system designers had only the partitions of hardware and software. And as we have seen, there traditionally was limited communication and interaction between the two spheres of design. The method of realizing an algorithm by means of a hardwired network of logic gates provides speed of execution and reliability. But it suffers from a lack of generality. It may also be more costly than a software approach. The historical solution to this architectural dilemma was to incorporate a sufficiently powerful but general machine instruction set as hardware resources which could be applied to algorithmic implementations by means of software development. While this method allows for much generality, it also generates much complexity at the user level and can be quite inefficient.

The influence of programming language theory on software development lead to the realization that machine hardware structured as close as possible to a source language is advantageous to the user.
Wirth (1971) noted that program construction may be perceived as a sequence of refinement steps in which program or problem statements are decomposed into more detailed instructions. This successive decomposition and refinement terminates when all instructions are expressed in terms of an underlying computer or programming language. Clearly, a machine whose underlying language (instruction set) closely approximates problem statements in its expressibility allows for an easy and efficient program specification. There have been investigations into the design of machines which directly execute high level languages without a translation of high level source code into low level machine code (Mullery 1964; Bashkow 1964; Bashkow, Sasson and Kronfeld 1967; Bloom 1973). There was also some progress towards implementing machine languages at a higher level so that the discrepancy between source code and machine object code was greatly reduced (Anderson 1961). The Burroughs B5500 and B6700 series machines with hardwired opcodes and stack mechanisms which closely reflect the structure of ALGOL are well known examples of this.

A progressive understanding of programming language theory and advances in technology have allowed the isolation and encapsulation of basic computing functions into integrated circuits. As a consequence, these hardware components have provided a means of redistributing system complexity so that more sophisticated systems can be built with less effort (Reigel and Lawson 1973). But there is still the problem of generality and flexibility. Machines that are designed to accept a particular high-level language as source code can still be inefficient and limited when applied to problems and computations outside the scope
of that programming language. And past experience seems to indicate that the universe of computer applications is ever expanding and that it is not practical to codify all forms of useful software features for implementing these applications into one general, all-purpose programming language. While hardware technology and economic considerations may develop to the extent that construction of several dispersed computers dedicated to particular functional tasks and programming languages becomes feasible, it is more likely that machines will be required to accept a variety of programming languages.

Such problems of machine design and function partitioning offer unique opportunities for microprogrammed assistance. The system units constructed as hardware or software interface at the intermediary microprogram level. Firmware implementations are capable of absorbing complexities from both the hardware and software directions of the spectrum (Reigel and Lawson 1973, p. 4). Microprogramming provides hardware designers with a more orderly and systematic procedure for control unit design. And it provides software designers with the flexibility of structuring machine architectures that are more adaptable to characteristics of problems which occur at their level of concern. Wirth's program specification perspective reveals problem statements and computer instructions as rungs on an "abstraction ladder" which proceed down from natural human language towards machine languages. The micro-commands which direct the machine to open gates, trigger flip-flops and perform various circuit functions occupy the bottom rung of this instruction ladder. Any type of instruction may be constructed from particular
sequences of these micro-commands stored in control memory as macro-instructions (Semarne and Porter 1961).

A system's functional mechanization consequently has three modes of implementation and each level in the distribution must be defended as to its existence. Distribution of a computing system's processing functions throughout the hardware-firmware-software mediums will be based upon the design goals and specifications of the virtual system. It is a postulate of this paper that microprogramming is itself a useful tool for assisting with the design decisions involved in such partitioning. The ebb and flow of technological considerations, economic constraints, programming theory and empirical evidence will produce systemic environments of diverse complexions with concomitant functional and medium redistributions. It will therefore be prudent to maintain a continuing examination of the hardware-firmware-software mix and appropriate justifications for specific virtual machine partition boundaries. The inherent power and flexibility embodied in the techniques of microprogramming provide unique opportunities for the analysis, synthesis, and experimental verification of new virtual machine designs.

A performance gain of decreased execution time is an attribute which is normally obtained by changing the medium of a function's realization from software to firmware to hardware, and it is theoretically possible to collapse an entire system configuration into hardware or some combination of hardware and firmware. Bell and his colleagues (1970, p. 668) estimated that a small computer with a cycle time of 1.0 microseconds possessing a 16 bit memory width would have the following characteristics for a floating point add operation (excluding data accesses):
programmed (software): 250 microseconds
microprogrammed (firmware): 25 microseconds
hardwired (hardware): 2 microseconds

A specific determination as to which of the three mediums would be most adaptable to a given system function will be related to the four principal criteria of speed, cost, reliability, and flexibility. Hardware represents the fastest but least general medium while software is of course the most general, but the slowest and often the least reliable. A basic dilemma that is encountered at the software and firmware level is that generality usually means less efficiency.

The principal attributes associated with hardwired implementations are of course speed and static structure. Sheer computational speed is usually the motivation for realizing a function in hardware. Complex computations that involve a large number of operations are most efficiently performed in hardware. Essentially any computer operation performed frequently and repetitively may be a candidate for hardware implementation (Falk 1974, p. 38). Critical functions identified within the virtual system hierarchy, be they associated with tasks of the operating system, the language processors or user programs, are subject to the possibility of a hardwired implementation. If such realizations can be justified as to their cost and static structure, then faster execution times can be obtained for the virtual system.

The static structure and stability of hardware makes the system more reliable in the sense that hardware is less subject to inadvertent modification. But in many situations this aspect of static structure
becomes a disadvantage. There are numerous environments (such as in the translation of different source languages and execution of various forms of object code) where one would like the available computational resources to assume a dynamic restructuring capability. Because such environments are often encountered in advanced software techniques and because of the trend in declining hardware costs, it has been noted that the optimum blend of rigid versus variable machine structure has shifted towards variability (Wilner 1972, p. 489).

Those functional realizations which must possess a certain degree of flexibility will represent candidates for the firmware medium of design. Flexibility is perhaps the most important advantage of firmware over hardware. Considerations of reliability and standardization are also characteristics which will contribute towards the selection of a microprogrammed approach. Under current and foreseeable technology, hardwired functions will continue to hold advantages of speed over microcode. But microprogramming introduces the prospects of greater machine standardization which may also imply greater system reliability.

It becomes feasible to construct a general-purpose, microprogrammable machine which can be specialized through its microprograms to perform any of several system functions. A system may then be composed as an integrated set of identical processors, performing different functions through microprogrammed specialization. For example one module could be dedicated to serve as a control processor, one as a peripheral processor and one as an application processor (Clapp 1972, pp. 19, 21). The reliability of such a system configuration is improved since it is
only the microprograms which distinguish one machine from another. Because the machines are identical, they are interchangeable and each serves as a potential replacement for one which has failed.

Another aspect of this standardization is that the implementation of new system functions is not dependent upon the development of new hardware. The basic, general microprogrammable machine represents an off-the-shelf hardware resource and it is only the development of requisite microprograms which needs to be accomplished for any new functional realization (Reigel et al. 1972, Davis et al. 1972). Further system reliability may be obtained through the development of microprograms for the purpose of hardware and software error detection, diagnostics and maintenance routines. Of course such routines could also be implemented in either hardware or software, but their requirement for flexible yet efficient processing make them good candidates for firmware (Cook, Sisson, Storey and Toy 1973; Flynn and Rosin 1971).

While the speed versus flexibility criterion is the determining factor in arriving at decisions between hardware and firmware implementations, matters of efficiency in terms of execution speed and memory requirements will be of principal concern in making choices between firmware and software mediums. Those software functions which are most suited to microcode replacement generally exhibit several distinct characteristics. As described by Clapp (1972, pp. 22, 23) a function which possesses one or more of the following attributes would be a good candidate for microcode replacement.
1. It is CPU-bound, rather than dependent on input/output transfer time.
2. It produces many intermediate results which are used in the processing and do not have to be preserved when the process is complete.
3. It is highly repetitive, either internally within the process or because it is frequently used as a part of some larger process, and consumes a significant portion of total execution time.
4. It is awkward to do with the existing instruction set and more natural to the microinstruction set.
5. The machine instructions used to perform the function have a high ratio of overhead time, spent on instruction fetch and address generation, to actual operation execution time.

Functions which are repeatedly executed are less efficient when performed by software. Within established cost dimensions, a high degree of complexity and repetitiveness may justify a complete upgrade into a hardware medium. However, less frequent execution and greater architectural flexibility will be more indicative of a firmware medium. Selecting a microprogrammed implementation for software tasks which are frequently executed can substantially decrease execution times. Typical increases in efficiency are due primarily to an elimination of multiple instruction fetches, subroutine linkages, excessive memory references, and redundant operations (Waldecker 1970, p. 78; Clapp 1972, p. 24; Abd-Alla and Karlgaard 1974, p. 803). Depending upon specific machine architectures, it may also be possible to accomplish more of the processing in parallel at the microprocessor level than with the set of machine instructions available. If a long sequence of software instructions can be consolidated into short microprograms, more efficient utilization of core memory is an additional advantage. The translation of high-level languages into concise intermediate languages for interpretation by microprograms is a good example of such code compactification.
The representation of programs in this form usually requires much less storage space than the typical expansion of required machine instructions (Clapp 1972, p. 36).

With the realization that practically any software feature within a computer system becomes suitable for microprogrammed implementations if it is executed with sufficient frequency, a list of tasks which are perceived as appropriate for microcode becomes quite lengthy. Several sources in the literature (Husson 1970; Wilkes 1969; Clapp 1972; P. M. Davies 1972; Cook and Flynn 1970; Van Der Poel 1962; Cook, Sisson, Storey and Toy 1973; Flynn and Rosin 1971; Reigel, Faber and Fisher 1972; Falk 1974; Jones and Merwin 1974) provide much discussion on actual and possible applications for microprogramming. The major means of classification is similar to that of software; that is the distinction between systems and applications (users) firmware.

Further categorization of microcode for systems applications includes the usual tasks associated with the functions of operating systems, language translators, diagnostics, input/output processing, and system interfacing. Specifically, the procedures which accomplish functions such as memory management, paging, garbage collection, indexing, address generation, stack operations, program relocation, interrupt processing, task scheduling, text editing, character string manipulation, parity checks, graphics display generation, and format checking, have been identified as features which are quite appropriate for microprogramming implementations. Microcoded versions of compilers, interpreters, lexical scanners, syntactic analyzers, and list processors are also very popular examples frequently cited.
Applications or **user-oriented firmware** normally involves the development of specialized operators tailored to the indigenous environment of the user. Common examples discussed in the literature include microprograms to perform the functions of table search, pattern recognition, block transfer, string, vector and array manipulations, sorting, floating point arithmetic, signal processing, Fourier transforms, and real-time processing. Another important area of user-oriented microprograms is concerned with the availability of microprogrammed simulators or emulators to allow the execution of user programs which reflect or require specific machine architectures. As the techniques of microprogramming become better developed, and facilities for the writing of microprograms become widely available, it is conjectured that the entire spectrum of system structure will be scrutinized and judged as to the appropriateness and susceptibility of microprogrammed realizations. Given sufficient resources, it is of course possible to directly code any type of applications program as microcode. It is the availability and expressive power of adequate microprogram creation aids such as micro-assemblers and high-level microprogramming languages that will to a large extent determine the domain of microprogrammed applications.

The most consequential and significant areas of microprogrammed applications are likely to be those endeavors directed towards the analyzation and subsequent modification of underlying machine structure. Because of the flexibility which is achievable with dynamic microcode, and because of the aspects of machine control which are embodied at the firmware level, it becomes possible to develop an iterative tuning
procedure which is very useful in the production of system architectures optimally designed for their environments. Data and statistics derived from the observation and measurement of the applicability of various system structures to specific types of computational problems has always been useful in the evolution of better machine designs. But the techniques of microprogramming offer unique opportunities to automate the evolutionary tuning process by providing greater data collection and machine restructuring resources. Motivation and characterization of microprogrammed techniques for the tuning process have been discussed in papers by E. W. Reigel (Reigel, Faber and Fisher 1972; Reigel and Lawson 1973).

The tuning process is composed of two stages. The first phase is concerned with the monitoring and tracing of applications programs to detect and collect a variety of statistics about the programs. This data is then used to identify and categorize certain program performance characteristics. For example, statistics that reveal the frequency of operation code usage, and sections of code which account for the largest percent of execution time are facts that are very useful. Using this information as feedback in the second phase of tuning, programming languages and machine designs which provide greater problem expressibility and efficiency may be constructed. Examples of improvements which are frequently indicated include the addition or deletion of instructions from the original instruction set, the combination of several machine instructions into one new instruction, the parallel manipulation of local registers and functional logic, or the composition of completely
new machine operations. Machine structure may also be modified to a
greater extent with the addition of stack registers, index registers,
or other general purpose registers, and appropriate alterations to
machine language formats.

The implementation of a tuning algorithm of this capability is
much easier and more efficient on a machine which possesses micropro-
grammed control mechanisms. To perform the monitoring functions, it is
only necessary to include a few additional microinstructions in each
macro or microprogram which defines a machine instruction for the
purpose of obtaining instruction set frequency counts and program pro-
files. In fact, much of the work on instrumenting compilers with respect
to program measurement could be directly applied at the microcode level.¹
Since the monitoring operations are introduced at a deeper level, and
because of the enhanced opportunity to perform micro-operations in
parallel, less instrumenting artifact would be introduced. The inter-
pretive nature of microprograms makes both static and dynamic profile
analysis possible. And once a suitable system structure has been syn-
thesized, the microcode which performs the collection of statistics could
be switched off for faster program execution. A diagram of the tuning
process as presented by Reigel et al. (1972, p. 722; 1973, p. 17) is
included in Figure 10.

¹ However, it is not yet clear what contributions towards the
problem of identifying and relating specific code to appropriate source
level statements can be devised by a microprogrammed approach.
Figure 10. Iterative Tuning Process.
By subjecting various procedures to a manual tuning process of similar nature, performance improvements of 10, 20, and 40 to 1 have been reported (Reigel et al. 1972, Reigel and Lawson 1973, Tucker and Flynn 1971). The first examples of improved code obtained from an automated analysis procedure were reported by Abd-Alla and Karlgaard (1974) and their improvements were on the order of four and eight to one.

**Programming Languages for the Specification of Microprograms**

The development of microprogramming languages, languages which allow the representation and preparation of microprograms, seems to parallel in many ways the design and implementation of conventional programming language translators. As in the evolution of traditional software translators (assemblers, interpreters, and compilers) for the codification and expression of algorithms, there have been produced a variety of software programming aids at different levels of linguistic complexity for transforming symbolic micro-source code into proper sequences of binary bit patterns. The bit patterns of course represent micro-object code to be entered and stored in control memory.

The level of program expressibility and degree of complexity for microprogrammed languages ranges from high-level machine independent micro-languages to low-level micro-assemblers and register transfer languages.

As characterized by Agrawala and Rauscher (1974), the spectrum of microprogramming languages consists of the following hierarchy:
The high-level procedure oriented microlanguages resemble conventional high-level machine independent programming languages and possess language constructs for compound expressions, data structure declarations and flow of control statements. Each microlanguage instruction may be translated into several microinstructions which are in turn composed of numerous micro-operations. Register transfer languages have fewer data types and generally have a one-to-one correspondence between language statements and machine microinstructions. The subcommands or micro-operations of a microinstruction are usually represented as simple statements. Micro-assembler languages express microinstructions in a mnemonic or symbolic form similar to traditional assemblers. In fact micro-assembler statements for vertical machines may appear exactly as symbolic assembly code, with label, opcode, and operand fields. For horizontal machines, micro-assembler syntax usually is composed of several additional fields, one for each possible micro-operation to be included in a microinstruction. Flowchart microprogramming languages are similar to micro-assemblers but are more topologically oriented. Sets of micro-operations are grouped together in boxes with links or flow lines connecting the boxes to indicate microinstruction sequencing (Agrawala and Rauscher 1974, pp. 821, 822, 830).
The usual dilemma between a microprogramming language's algorithmic expressionality and automation, and the efficiency of the microcode it generates is encountered. Research and development into the techniques for the construction of high-level microprogramming languages has only recently been pursued. The classical debate between the efficiency of low-level translators and the ease and labor saving aspects of high-level translators has been resumed in this area of investigation. Resolution in favor of one philosophy over the other will probably not occur as quickly in the specialty of microprogramming as it did in the broader environment of software development. The reason for this is mainly economic and has to do with the number of people that will eventually become involved with the discipline and practice of microprogramming. And this question itself has not yet been decided. It is often argued that the philosophical emphasis of the art of microprogramming should be directed towards the absorption of system complexity at a lower level so as to isolate users from much of the detailed entanglements of system organizational features. The ultimate purpose of this approach is to have fewer people involved at the level which exhibits the most complexity (Reigel and Lawson 1973, p. 4). If the number of practitioners concerned with the techniques of microprogramming is relatively small, then low-level translators capable of generating extremely efficient microcode will probably be the prevalent means of microprogram preparation.

However, it is also a possibility that since the user (or at least a certain species of user) is the person with the most direct
knowledge of the applicable environment and appropriate machine architecture, the facilities for the representation and preparation of microprograms should be accessible at the user level. It has been speculated that the number of users who will become authors of microprograms will consequently increase (L. H. Jones 1973, p. 17). A widespread interest and involvement in microprogramming procedures would then exert a demand for higher level microprogramming languages.

It should also be noted that the degree of automation which characterizes the software resources available for microcode translation is significantly related to the architectural features of the microprocessor. Specifically, under current microprogramming language strategies, those microprocessors which exhibit vertical structures are usually provided with the higher level language constructs for their microcode translators, while horizontal microprocessor structures are generally accompanied with low-level micro-assemblers or register transfer translators (Rauscher and Agrawala 1973, p. 52; Agrawala and Rauscher 1974, p. 831). The reason for this is ascribed to the fact that it is extremely difficult to devise higher-level programming language constructs for translators which are capable of analyzing the inherent complexity and parallelism of horizontal microinstructions. As previously discussed, decisions which result in a horizontal microprocessor design are normally made with the intention of obtaining highly efficient and optimized system structure. In order to take advantage of such system resources it has been necessary to rely upon low-level translators and the expertise of the microprogrammer.
Progress towards the application of optimization techniques for the production of efficient horizontal microcode from high-level source code has been reported in a paper by Ramamoorthy and Tsuchiya (1974). By including stages for micro-operation concurrency, timing analysis and optimization to the standard phases of syntactic analysis and semantic synthesis, Ramamoorthy and Tsuchiya were able to construct a machine independent ALGOL like microprogramming language compiler which generates sequences of compact, efficient, and, where possible, parallel micro-operations from high-level source code. An illustration of the compilation procedure, as described by Ramamoorthy and Tsuchiya is depicted in Figure 11. Although good results were obtained, the elaborate analysis during compilation consumes a large amount of translation time (Ramamoorthy and Tsuchiya 1974, p. 798). Longer compilation times and greater development costs for translators of increased complexity are added expenses which can be expected to occur when optimization of microcode is attempted. The associated ease of programming in a higher-level language and the consequent savings of man-hours required for microprogram specification may of course justify the increased expenditures.

Until the relationship between the generation of efficient microcode from microlanguage compilers and the productivity of microprogrammers is better established, high-level procedure oriented microprogramming languages will probably be limited to vertical microprocessors. And since the organization of microprocessor architecture (vertical or horizontal) is motivated by considerations of the functional objectives to which the system will be applied, the type of
Figure 11. High-level Microlanguage Compiler for Generation of Efficient Horizontal Microcode.
software facilities available for the construction of microprograms will largely be dependent upon projected system functions, that is, upon the design characteristics of the operating engine. Just as it has been argued that one should approach machine organization from the perspective of language-directed machine design, it is conjectured that the same philosophy would be beneficial at the microprocessor level. The sequence of design tasks leading towards system realization then becomes:

1. Determine functional applications of virtual system and specify suitable resources for required operating engine.
2. Select appropriate microprogramming language for representation and preparation of microprograms.
3. Derive efficient microprocessor structure to interpret microcode for the generation of proper control signals to direct operating engine.

Questions of research related to appropriate machine architecture may therefore be conceptualized at two hierarchical levels. There is the high-level realm of architectural designs which are optimized for the execution of virtual machine algorithms. And there is the lower level of microprocessor structure designed for efficient interpretation of microprograms. Problems at the virtual applications level will be concerned with the structuring and control of computing resources for the execution of algorithms and procedures of the following type: programming language translators, numerical analysis, data base processing, information retrieval, sorting, simulations, game playing, natural language processing, list processing, robotics, pattern matching, and
real time control. Questions at the lower level will be involved with matters such as the manner in which microsequencing is accomplished, the status tests and conditional relations that direct microsequencing, the availability of structured programming constructs at the microinstruction level, and the relationship of microinstruction sequencing functions to the ease of writing correct microcode and the size of control memory required. An interesting extension of this line of thought is the question of whether or not hardwired stack resources should be included in the microprocessor organization.

Microprocessor Architecture and Implications

Many of the microprocessors for contemporary microprogrammable machines are specifically structured towards the support and interpretation of machine instructions sets which compose the virtual or emulated system machine language repertoire. As explained by Agrawala and Rauscher (1974, p. 828), machines of this type do not provide a compatible resource base for other microprogrammed applications. The conditional branching and sequencing capabilities of the consequent microinstruction sets are frequently very awkward to use. The lack of structured programming primitives at the microinstruction level makes it difficult to implement such concepts at the programming language level (L. H. Jones 1973, p. 20). The substantial possibilities of parallel processing by means of grouping parallel sets of micro-operations is often not completely exploited. In general the relationship between primitive microprocessor facilities most suitable to the different
aspects of microprogrammed applications requires more study. Research projects such as the one discussed by R. G. Barr et al. (1973) are greatly warranted. As illustrated in that paper, the collection of empirical data about microprocessor characteristics through the observation and measurement of microprocessor resource utilization, microinstruction bit utilization, the number of micro-operations grouped into single microinstructions, and microinstruction frequency and type analysis can yield valuable results. Questions about the degree of parallelism or concurrency of micro-operations which represent an optimal resource base for efficient language interpretation, and appropriate sets of micropredicates or status conditions for microbranching are other examples of areas for further investigation.

As additional data on the changing aspects of various architectural philosophies are accumulated by the iterative tuning process previously discussed, it should be anticipated that the extensive reliance upon general purpose machine instruction sets as a programming vocabulary will be reduced. Rather than having large and complicated sets of machine instructions which the programmer must remember as a generalized vocabulary to be applied to many different situations, a small set of powerful and flexible commands will be available for each specialized application. In addition to the direct execution of high-level languages, it is likely that microprogramming will result in the development of specialized instruction sets tailored to perform, for example, the functions of language translation, string and list
processing, pattern recognition and associative processing. The functions of programming language translation (lexical scanning, syntactic parsing, and code generation) seem particularly suited to firmware implementations and indeed much contemporary work has been directed towards the realization of microprogrammed machine structures for the FORTRAN, COBOL, ALGOL, SNOBOL, LISP, and APL programming languages (Melbourne and Pugmire 1965; Rosin, Frieder and Eckhouse 1972; Wilner 1972; Hassitt, Lageschulte and Lyon 1973; Weber 1967; Deutsch 1973).

Adoption of these propositions in their entirety will culminate in a machine structure something on the order of Figure 12. An architectural trend of this sort would seem to be appropriate, considering the capabilities and constraints of contemporary technology and economics.

In the past, the traditional approach to computer design involved a concentration on the development of hardware for the fast execution of numerical algorithms. The major emphasis was directed towards the efficient execution of procedures expressed in terms of a general machine instruction set. After hardware of this sort was generally available, the problem of specifying and writing programs began to receive more attention. Programmers attempted to have the machine accomplish those programming tasks which were characterized as mechanical and repetitive.

1. This has been a much touted use of microprogramming, but to date no significant examples to indicate the advantage of this approach can be given. It is still the author's opinion that instruction set development along this line of thought would be beneficial.
Figure 12. Microprocessor System Configuration.
In essence, they wanted to shift the burden of bookkeeping and housekeeping functions (which characterizes a large portion of any programming effort) to the machine in order to allow their concentration on the creative aspects of programming. This effort resulted in the development of high-level procedure oriented programming languages which removed much of the laborious housekeeping and standardization tasks from the concerns of the applications programmer.

As a direct consequence of automating some programming requirements, larger portions of computer time were expended during the translation of high-level procedures into machine code. These transformation operations were of course performed with the same set of registers, addressing mechanisms, interrupt schemes and arithmetic logic units that were originally designed for the expression and execution of numerical algorithms. As the application and theory of programming languages became more advanced, instructions which were appropriate for the functions of language translation (push, pop, translate and test, and test mask instructions for example) were included in the machine instruction sets of several second and third generation computers, but the same computing resources were used for both the translation to and execution of machine code. Essentially, machine designs intended primarily for the execution of code expressed in the form of arithmetic operation codes and one or two operands, were being extended into an applications area in which the programs exhibited a very different topological structure. Because of the state of hardware technology and associated costs, this situation was entirely understandable.
But continued advances in the abilities of hardwired logic units and a downward trend in hardware costs portend alternative solutions. It is now feasible to provide separate hardware resources for the functions of translation and execution. Furthermore, the control and informational bindings between the registers, logic units and buses of the translation (T-machine) and execution (E-machine) resources do not have to be solidified into permanent relationships. The flexibility of microprogramming can restructure system organizations for either application.

Much has been projected for the implications of microprogramming. Actual realizations and implementations of the technique have been slow in development, limited in availability, and restricted in perspective. But the philosophy of microprogramming does represent a new style of approach to the problems of system design and development. As a technique, it exists as an extremely useful research tool. Its flexibility and power seem particularly adapted to experimental design work directed towards the investigation of appropriate hardware, firmware and software functional distributions. The most important aspect of a microprogrammable processor is the concept of delayed binding. By delaying the association of relationships between certain system elements from machine design time until microprogram execution time, it is possible to tailor the organization of system resources to fit the nature of the problem to be computed at program execution time. This redefinable structure also permits conceptual developments from the topics of automata theory, programming language theory, linguistic theory, and the theory of data structures to properly influence the specification of machine
architecture. Architectural changes which are motivated by a better understanding of the primitives that are conceptually employed in an applications process will lead to a better mapping of requirements onto computing resources (Clapp 1972, p. 30).

Comparisons between the activities of programming and microprogramming usually indicate a conceptual similarity. If the microprocessor is vertically structured and/or possesses a relatively high-level micro-language, it may indeed appear that one is programming in the usual sense. Although much of the effort involved in the creation of microprograms is isomorphic to the problem of general software development, there is an important distinction about the characteristics of the end results. Microprogramming is concerned with generation of control signals and orchestration of processor resources. The level of resource allocation which occurs in the domain of microprogramming is one level deeper than that of traditional programming.

The target of either assembly or high-level language programming is of course the specification and generation of a correct sequence of machine code. The necessary control signals which orchestrate the machine's functional resources are then produced as each machine instruction is decoded and interpreted by the hardware. Since the machine instruction set represents a fixed subset of relationships between computer resources, the static set of possible control signals is often found lacking. The deeper level of computer resource configuration available with a microprogrammable processor increases the set of possible control signal propagations and is, as a result, the source of
the widely heralded flexibility. Associated with this flexibility are two principal costs; increased programming complexity, and slower execution times. Further developments in the optimization powers of high-level microprogramming languages should reduce the effect of the former, while new technological developments may reduce the effect of the latter.

Conclusions

This discourse has presented a description of the concepts and techniques embodied within the subject of microprogramming. The definition of common terminology and classification of microprogramming themes into three categories proceeded from a review of the literature. Contributions towards a theory of microprogramming adapted from formal automata theory were discussed. The formalization produces a structural abstraction of microprogrammable computers as a hierarchy of automatons. Procedures for minimization and verification of microprograms are benefits which will derive from the continued development of such a theory.

Determination of a suitable microinstruction format for flexible and efficient management of a machine's primitive resources was identified as the principal objective in the design of microprocessors. A taxonomy of microinstruction formats however revealed the distinctions between two philosophical approaches to microprocessor design. A horizontal philosophy will seek to optimize the number of resources which can be simultaneously controlled by the microprocessor, whereas a vertical solution will concentrate upon the specification of a small but powerful set of microinstructions which can be efficiently applied in
algorithmic procedures. The design goals implied by these two general approaches are realized by selecting different methods (from the spectrum of microprocessor design strategies) for the propagation of control signals from microwords, the generation of control store addresses, and the manipulation of status information. Specifics of monophase/polyphase microword interpretation, bit pattern/control function assignment, and microinstruction sequencing were discussed, with the development of a pedagogic microprocessor model used to illustrate the entailed concepts.

Relationships between the type of microprograms required to effectively direct resources of the operating engine, the appropriate microprocessor designs for interpretation of that microcode, and characteristics of microlanguages for coding microprograms were also examined. It was noted that since the actual structure of the operating engine exerts considerable influence on the type of microprocessor design required, the environment which is foreseen as the virtual system's domain of operations will in large measure determine the approach towards microprocessor construction. This in turn will influence the type of microprogramming language which can be used for the expression and translation of microcode. Horizontal microprocessors will normally require low-level micro-assembler languages in order to produce the highly parallel and efficient object microcode which is desired. The simpler, sequential characteristics of vertical microinstructions allow high-level, procedure-oriented languages to be used in the generation of microprograms for vertical microprocessors.
The concept of redefinable structure derived from a delayed binding of physical structure and relationships among system components was characterized as a most important consequence of the microprogramming technique. This delayed binding results in a larger set of possible control signal configurations under the direction of a programmer. It was suggested that the domain of microprogramming applications exists across the entire spectrum of subcomponents for virtual system configuration. That is, microprogrammed solutions could be integrated with the traditional media of hardware and software for the realization of operating systems, language processors, and user programs. And it was further suggested that a very important application of the microprogramming technique would be as a mechanism for the determination of appropriate hardware-firmware-software mixes. An iterative process for the measurement of architectural features as a means of collecting data to direct the construction of more effective designs was contemplated.

The underlying machine structure implied by the perspective of a microprogrammable processor split questions of research on machine architecture into two hierarchical levels. There is a correspondence between the type of questions to occur at each level and whether they are about the operating engine or about the control engine. Investigations into machine resources appropriate for efficient execution of application algorithms (problems of numerical analysis, information retrieval, list processing, language translation, etc.) are directed towards the proper organization of the operating engine. A specific research objective at this level, for example, would be the determination
of an optimal resource base in terms of concurrent micro-operations for programming language interpretation. Studies of good microprocessor facilities (microinstruction sequencing, conditional tests, bit utilization, parallel micro-operations, etc.) for the interpretation of microcode to drive the operating engine configurations which result are essentially about the control engine. Important questions regarding this type of research include the matter of providing structured microprogramming constructs, recursive microprogram calls, hardware microprocessor stacks, and appropriate sets of micropredicates to control microbranching.
SELECTED BIBLIOGRAPHY


