A COMPREHENSIVE SUPPORT SYSTEM FOR MICROCODE GENERATION

Juan Linares

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfilment of the Requirements

for the degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

August 1982

ABSTRACT

A COMPREHENSIVE SUPPORT SYSTEM
FOR MICROCODE GENERATION

Juan Linares

This thesis examines the process of microcode production. A survey of current microprogramming techniques is presented to identify specific problems. The problem of microcode production is shown to have several interrelated components, namely: machine description, microprogram specification, microprogram verification, and microcode optimization.

A comprehensive solution to this problem is proposed with the design of a Comprehensive Microprogramming System (CMPS). The system presents solutions to specific problems of microcode production while maintaining the relationships among the various phases of microcode production.

The two main components of the system are the Hardware Abstraction Language (HAL) and the Alternative Based Microprogramming Language (ABMPL). HAL is used to describe the microarchitecture of a computer and ABMPL is used to specify the microprograms. The features of both of this languages are discussed extensively throughout the thesis and some examples are provided. For both of this languages prototype compilers have been implemented and some practical experiences have been carried out in the SEL32/75.

ACKNOWLEDGEMENTS

I would like to express my sincerest thanks to my thesis supervisor Dr. Terrill Fancott, who enthusiastically guided me through this thesis. His numerous clarifications and suggestions have proven invaluable during the course of this project. I also would like to thank Dr. I. Greenshields, who as my initial supervisor stimulated my interest into microprogramming, and outlined the general direction of this research.

I also would like to express my gratitude to Dr. W. Jaworski for having taught me a new and fascinating way to create programs. His ideas on program development techniques have profoundly affected my ideas on software development. The concepts involved in his Alternative Based Language have contributed heavily towards the development of the microprogramming techniques presented in this thesis.

My thanks also go to D. Hargreaves, M. Duarte, and G. Mack, who at various times helped me with the implementation of microprograms for the SEL32/75 here at Concordia University. I also would like to thank my employer LAGOVEN S.A., of Caracas Venezuela, whose financial support of my studies here at Concordia University made the completion of this thesis possible.

I dedicate this thesis to my wife Lilian whose patience and moral support helped me through the completion of it.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Contemporary microprogramming research within the academic community is principally concerned with the design and implementation of reliable systems for the generation of compact and correct microcode. Shriver & Lewis in a recent paper on microprogramming stated this succintly as follows: "contemporary microprogramming is concerned with firmware engineering, i.e., the reliable implementation of correctly functioning microprograms" [SHRI81].

This interest has not yet, however, yielded good tools for the production of microcode. In some cases users have been concerned with applications of microprogramming, and have disregarded the development process [SHRI81]. In other cases researchers have attempted to borrow ideas from software engineering to solve problems in firmware engineering which "are not ordinarily encountered in the software domain" [DASG80].

1.1) BRIEF HISTORICAL SURVEY OF MICROPROGRAMMING

Microprogramming was originally proposed by M. V. Wilkes in 1951 as a systematic alternative to the ad hoc method of designing the control system of digital computers prevalent at the time. Besides being a more structured approach, it introduced a large degree of flexibility in the design, implementation, and maintenance of the instruction set of a computer [WILK69].

By now Wilkes' matrix model, shown in figure 1.1, has become a classic in computer science. In this model at each clock pulse matrix 'A' would emit signals affecting the various gates in the CPU and matrix 'B' would supply information to determine the address of the next microinstruction. This scheme of logically dividing the information in a microinstruction into functional information (microoperations with the hardware) and sequencing information (addressing, tests, etc.) has remained basically unchanged until now [MICK77].

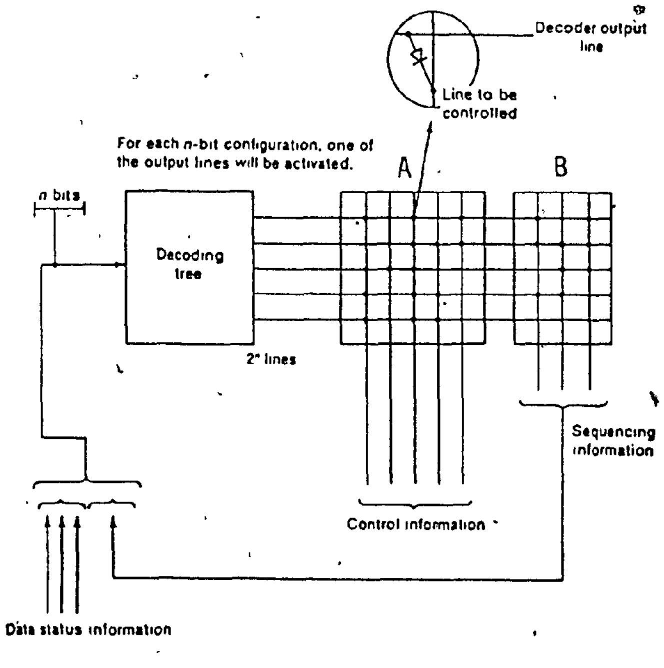


Figure 1.1 Wilkes' Matrix (from FLYN80)

The next stage in the evolution of microprogramming was determined by the interest in microprogramming as a means of designing a range of computers of differing power but with compatible instruction sets. The best example of this is IBM's 360 series in which all machines were at least upward-compatible. In this series all but the largest computer then announced (model 70) had microprogramming based on ROM [STEV64].

This contributed to the development of hardware emulation as an important research topic. Tucker defined an emulator as a package that includes special hardware and a complementary set of software routines [TUCK65]. Emulation does not therefore imply the implementation of an entire instruction set in a microprogram. An emulation package is the result of a careful comparative analysis of the target and host architectures. Particular implementation decisions are based on the study of the difficulties and advantages of a software versus microprogrammed implementation. A machine instruction may be microprogrammed if its software implementation is too difficult, or too inefficient. Another reason might be that a particular instruction is used so often as to be worth the effort of microprogramming it.

The latest phase in the evolution of microprogramming is characterized by the appearance of user-microprogramable machines which provide the tools to carry out research on the various aspects of microprogramming. Advances in integration technology have led to the appearance of powerful microprocessors which have given great impulse to microprogramming, since these usually employ microprogrammed control units. Shriver & Lewis in a recent paper on microprogramming [SHRI81] write: "the LSI processor forced designers of multilevel interpreters to reevaluate microprogramming. Instead of reducing costs by simplifying the design and increasing flexibility, microprogramming became a way to increase the regularity of structure within an LSI chip".

To this day the goal of widespread user-microprogramming has remained elusive due to the lack of appropriate firmware tools. Their development will have profound effects on the field of computer science.

1.2) A COMPREHENSIVE APPROACH TO MICROPROGRAMMING

The problem of generating good microcode will not be solved simply by the introduction of high level languages and their compilers. This approach has been overemphasized in the current microprogramming literature to the point of obscuring other issues in the problem of user microprogramming. The specification of microprograms in a high-level notation should be a consequence of a comprehensive solution to the overall problem of user microprogramming. This solution must address the problems of machine description, optimality of microcode, microcode correctness, and user convenience. Shriver & Lewis, when referring to the ideal microprogramming tool, wrote: "work toward this hypothetical microprogramming tool has been thwarted by false starts and movement in tangential directions. One of the most obvious tangents has been the invention of new languages for microprogramming. These languages often incorporate ideas borrowed from software engineering. Data abstraction is the most frequently borrowed idea. At their worst, many high level languages incorporate low-level host machine features that we are attempting to hide by using a high level language in the first place" [SHRI81].

Microprograms are very intimately related to the microarchitecture in which they operate. Microarchitecture means all the hardware items which can be referred to or operated on by a microprogram. Dasgupta defines it as " the nature of the interface visible to the microprogrammer" [DASG80]. Given the above relationship any proposal to generate microcode must be concerned not only with the structure of the language used to specify microprograms, but also with the microarchitecture in which these are executed.

The central proposition of this thesis is the design of a comprehensive microprogramming system which addresses the various aspects of microcode generation, in particular those related to microprogram specification and microarchitecture description. Baer, when defining a microprogramming system writes: "the generation of a microprogram (microcode) is the result of a process which receives as input descriptions of the algorithm to be executed and of the hardware resources needed to perform and sequence the operations" [BAER80].

Baer proposes the system of figure 1.2 and writes that "the automated translation of the microengine (microarchitecture) description into templates of microinstructions is a problem which is far from being solved" [BAER80].

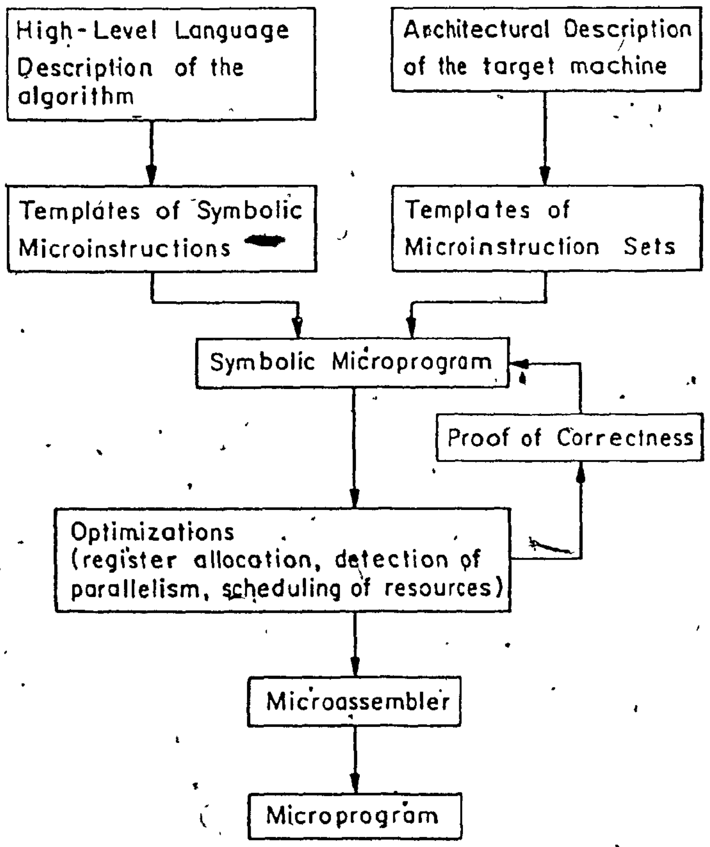

High-Level Language Description of the algorithm
Architectural Description of the target machine
Templates of Symbolic Microinstructions
Templates of Microinstruction Sets
Symbolic Microprogram
Proof of Correctness
Optimizations (register allocation, detection of parallelism, scheduling of resources)
Microassembler
Microprogram

Figure 1.2 Baer's Microprogramming System (from BAER80)

The system proposed in this thesis has a structure similar to Baer's but addresses the problem of machine description by introducing the concept of hardware abstraction, analogous to the abstract data type concept. This abstraction constitutes a high-level language image of all user accessible storage structures, and operations available to the user. Microprograms will be specified in a high-level language which uses the operations and data items specified in the hardware abstraction data type. The logical structure of the language is based on the Alternative Based Language technique [JAWO80] since it allows the abstraction of machines and programs, and is therefore in line with the overall strategy of this work.

1.3) THE ORGANIZATION OF THIS THESIS

Chapter two of this thesis examines the current state of microprogramming technology, in both hardware and software aspects, and the problems currently associated with microprogramming.

Chapters three, four, and five are used to develop the central proposition of this thesis. Chapter three deals with the description of computers in a way which is accessible and meaningful to application programmers. The concept of hardware abstraction is discussed and a hardware abstraction language to describe a computer is presented. In chapter four a practical example is given with the description of components and operations in the SEL 32/75 as hardware abstractions.

Chapter five is dedicated to microcode generation. The comprehensive microprogramming system is presented in its various aspects. The generation of microcode through an alternative based microprogramming language is discussed. Finally in chapter five some practical examples are presented with microprograms implemented through the system for the SEL 32/75.

It should be noted that the hardware abstraction language and the alternative based microprogramming language are complementary components of the system. One has no meaning without the other. In this thesis we have chosen to present the hardware abstraction language first, and then to introduce tha alternative based microprogramming language. This order of presentation was chosen to emphasize the hardware aspects of microcode generation.

For those whose background is principally concerned with software it might seem more appropiate to approach the two system components in the reverse order. In such a case chapter five may be read before going to chapters three and four.

In chapter six the conclusions of the study are presented. The feasibility of the system is discussed as well as its ability to solve some of the problems currently associated with microprogramming.

CHAPTER 2: CURRENT ASPECTS OF MICROPROGRAMMING

The aim of this chapter is to familiarize the reader with the current status of microprogramming technology in its various aspects. Mick defines a microprogrammed machine as "one in which a sequence of microinstructions is used to execute various commands required by the machine. If the machine is a computer, each sequence of microinstructions can be made to execute a machine instruction. All of the little elemental tasks performed by the machine are called microinstructions" [MICK77]. Microinstructions can be further decomposed into microoperations which are hardware based and are the most primitive actions that can be effected in the context of the microarchitecture.

The study of microprogramming hardware may be divided into three main areas which are related, but complex enough to deserve independent consideration. These are: control storage organization, microinstruction format and microinstruction sequencing.

2.1) CONTROL STORAGE ORGANIZATION

Control storage refers to a store from which microprograms are executed. This does not imply that control storage is distinct from main memory, although that is often the case. Most microprogrammed computers store microprograms in a smaller but faster memory, but there are some exceptions such as certain models of the IBM 360 series and the Burroughs B17ØØ in which microprograms are executed from an area of main memory [DASG79].

Dasgupta defines control storage as "a store in which microprograms reside, and whose organization and design is determined solely from the viewpoint of microprogramming" [DASG79]. One of the major disadvantages of microprogramming compared to hardwired control, is the time involved in fetching microinstructions from control storage. This factor can be made insignificant by appropriate implementations of control storage and microinstruction execution; hence the importance of control storage organization.

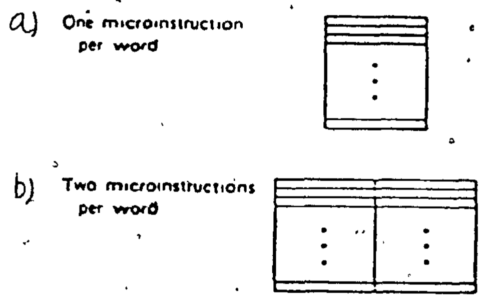

a) One microinstruction per word
b) Two microinstructions per word

Figure 2.1 Array Organization (from RAUS8Ø)

Control storage can be logically organized in several ways. The simplest and most common structure is the ordinary memory array with one microinstruction per word [fig. 2.1a]. A variation of this form is to increase the size of the microword in order to accommodate two microinstructions. The advantage of this is that fewer memory references are required since two microinstructions can be accessed simultaneously [fig. 2.1b].

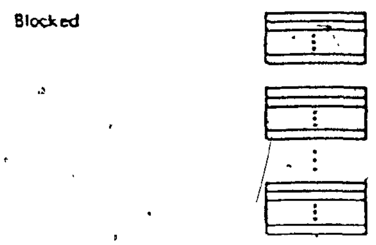


Figure 2.2 Block Organization (from RAUS80)

Another form of organization is in blocks [fig. 2.2]. In this scheme there are two types of addresses, one of microinstructions in the same block as the current microinstruction and the other of addresses of other blocks. As a result, addresses of microinstructions in the same block are shorter than in a non-blocked structure. This organization is efficient if the microprogram can be organized in branch-free blocks, each block processing a certain specific machine instruction. This is usually not an easy task, but with appropiate software it may be achieved.

In the split structure [fig. 2.3] we have two different storage units with different word sizes. The unit with shorter word length contains microinstructions which utilize very few resources or initiate the execution of a microinstruction in the other unit. The second storage unit has many more bits per word and therefore can exercise more control over machine resources. This organization can be very efficient if most microinstructions executed are of the short type or if many of them reference the same long microinstructions.

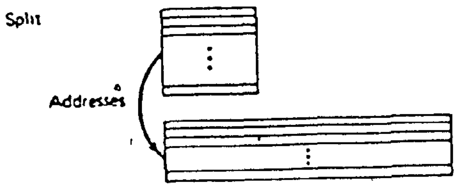


Figure 2.3 Split Organization (from RAUS80)

In a two-level organization the instructions in the lower level memory unit interpret those in the upper level [fig. 2.4]. This is called nanoprogramming and is conceptually similar to microprogramming, however this organization provides flexibility in the design of microinstructions as well as machine instructions. Machines with this organization are usually used for research into microprogramming. This organization of control storage is used in the QM-1 machine from Nanodata Corporation.

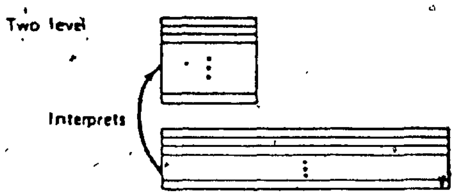


Figure 2.4 Nanoprogramming (from RAUS80)

For the purpose of this thesis, practical experiments will be carried out on the SEL32/75 computer available here at Concordia University. The SEL is a 32-bit user-microprogrammable computer oriented towards real time applications, but it can also handle general batch applications. It is therefore important to describe the particular microprogramming technologies which the SEL uses.

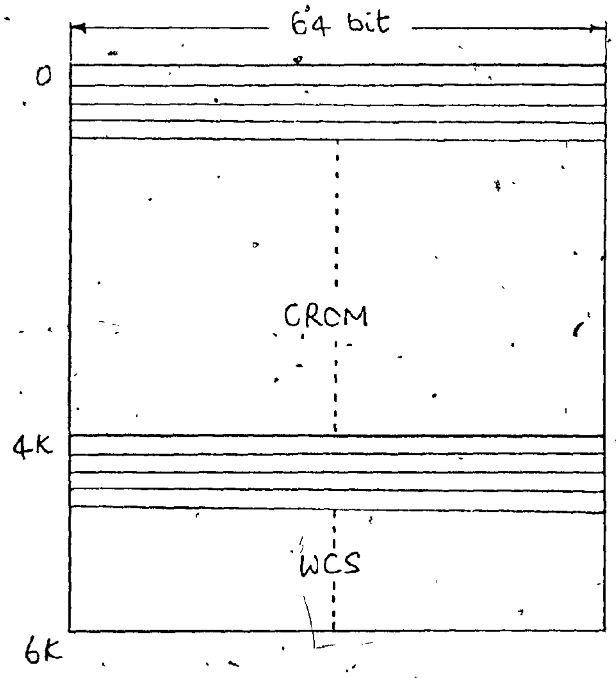


Figure 2.5 The SEL's Control Storage

The control storage of the SEL 32/75 [fig. 2.5] is of the single-word array type. It is organized as an array of 64-bit words, each word containing one microinstruction. The first 4K words consist of ROM which contains, among other things, the microprograms to interpret the basic instruction set of the SEL 32/75. The rest of the control storage is used as a Writable Control Storage (WCS) which is installed in increments of 2K upto 8K. The WCS is made up of RAM and it is used for user-microprogramming. The WCS is viewed as an extension of the ROM control storage and therefore there is continuity in the range of addresses [SEL1].

2.2) MICROINSTRUCTION FORMAT

A microinstruction is merely a string of bits whose meaning (use) is determined by the decoding hardware. Of primary interest in the design of microinstructions is: the number of resources each microinstruction controls. In this respect microinstructions are classified as vertical or horizontal [RAUS80] although these appellations refer to the extremes of a broad spectrum.

Vertical microinstructions effect single operations such as LOAD, STORE BRANCH etc. They often resemble machine language instructions containing one or more operands.

Horizontal microinstructions, in contrast, control many resources which may operate in parallel. A microinstruction might control, for example, the simultaneous and independent operation of the ALU, input and output to main memory, conditional next address generation etc. Horizontal microinstructions have the potential advantage of efficient hardware utilization, but the optimization process is a difficult task.

For both vertical and horizontal formats, there are several ways to implement the executions of microoperations.

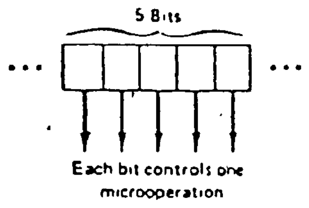


Figure 2.6 Direct Control (from RAUS8Ø)

In DIRECT CONTROL, every bit in the microinstruction represents a microoperation and is converted into a control signal which inmediately and directly controls a certain machine resource [fig 2.6]. In the extreme horizontal case there would be one bit for every microoperation possible in the machine. This scheme provides flexibility, but results in large microwords and a waste of storage. It is of historical importance only since it corresponds to Wilkes' model and it is rarely used.

In RESIDUAL CONTROL a group of set up registers are used to control resources [fig. 2.7]. Each register controls a particular resource and microinstructions may then replace or alter the value in one or more registers. The set up registers may be manipulated by a sequence of vertical microinstructions, yet they simultaneously control several resources as do horizontal microinstructions. In situations where a certain resource performs the same operation repeatedly this scheme can provide substantial savings in control storage use.

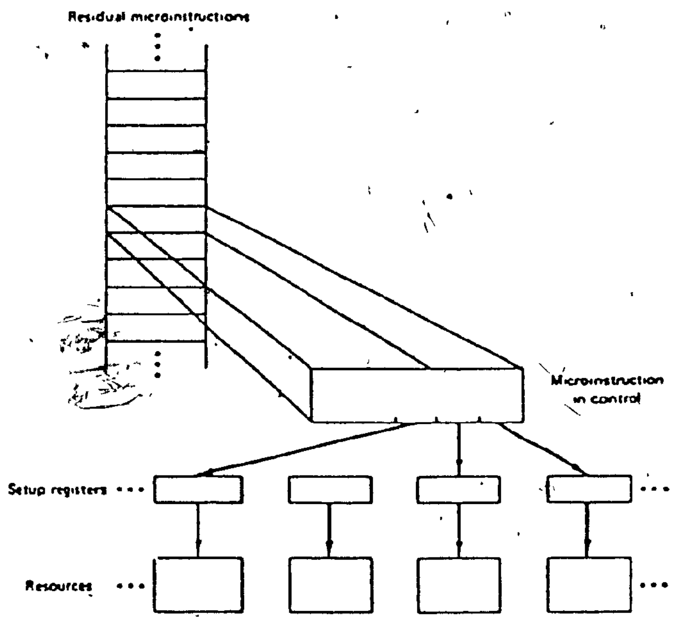


Figure 2.7 Residual Control (from RAUS80)

In MAXIMAL ENCODING each microinstruction is considered to be a unique state of a microword. If there are n possible states (microinstructions), then we can represent each microinstruction with a code using log2(n) bits. This scheme is most effective in minimizing microword length, but if the number of states (microinstructions) is large, a significant decoding delay may be introduced [fig. 2.8]. Furthermore this scheme lacks flexibility, because the addition of new microinstructions may result in hardware modification.

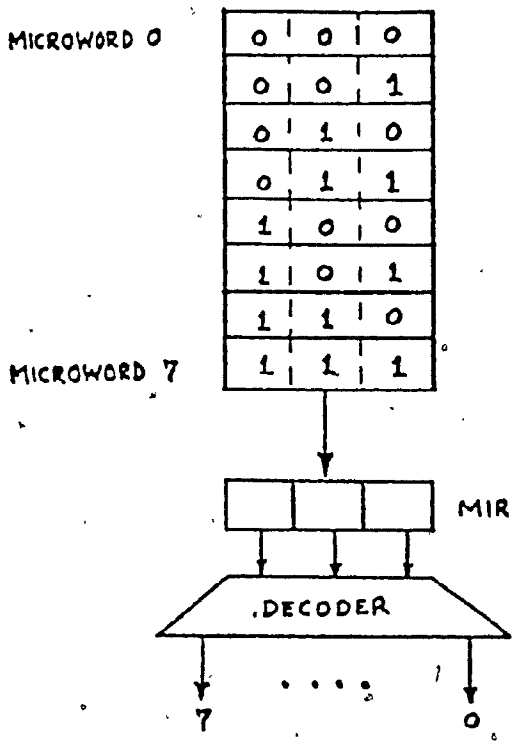


Figure 2.8 Maximal Encoding (from DASG79)

In MINIMAL ENCODING, microoperations on mutually exclusive resources are grouped into a sets represented by fields within the microinstruction. Its simplest form is called single level or direct encoding [fig. 2.9a]. In two-level encoding the meaning of a field is made to depend on the value of a control field within the microinstruction [fig. 2.9b]. This is sometimes referred to as bit steering. Another arrangement of two-level encoding is called format shifting, where the value of a field depends on the machine state indicated by some status registers.

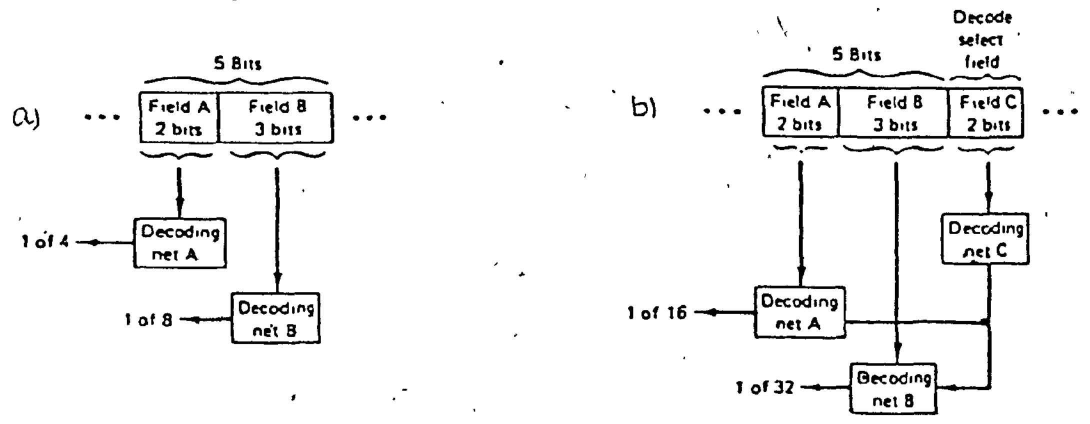


Figure 2.9 Minimal Encoding (from RAUS80)

The most widely used scheme is a combination of the various forms of minimal encoding. This the scheme used by the SEL 32/75. The format of the microword for the SEL is shown in figure 2.10 . Certain fields such as the T field are used as control fields to determine the meaning (use) of other fields such as the M, X and P fields. Other fields such as the '+' field do not depend on any other field for the purpose of decoding.

2.3) MICROINSTRUCTION SEQUENCING

The microinstruction sequencing mechanism is a great source of variability among microprogrammable machines, since they all have different and often inconvenient addressing mechanism [PERS77]. Jones has studied the possibility of implementing high-level structured control constructs (REPEAT UNTIL, WHILE DO, etc.) using the available sequencing mechanisms and has found these lacking

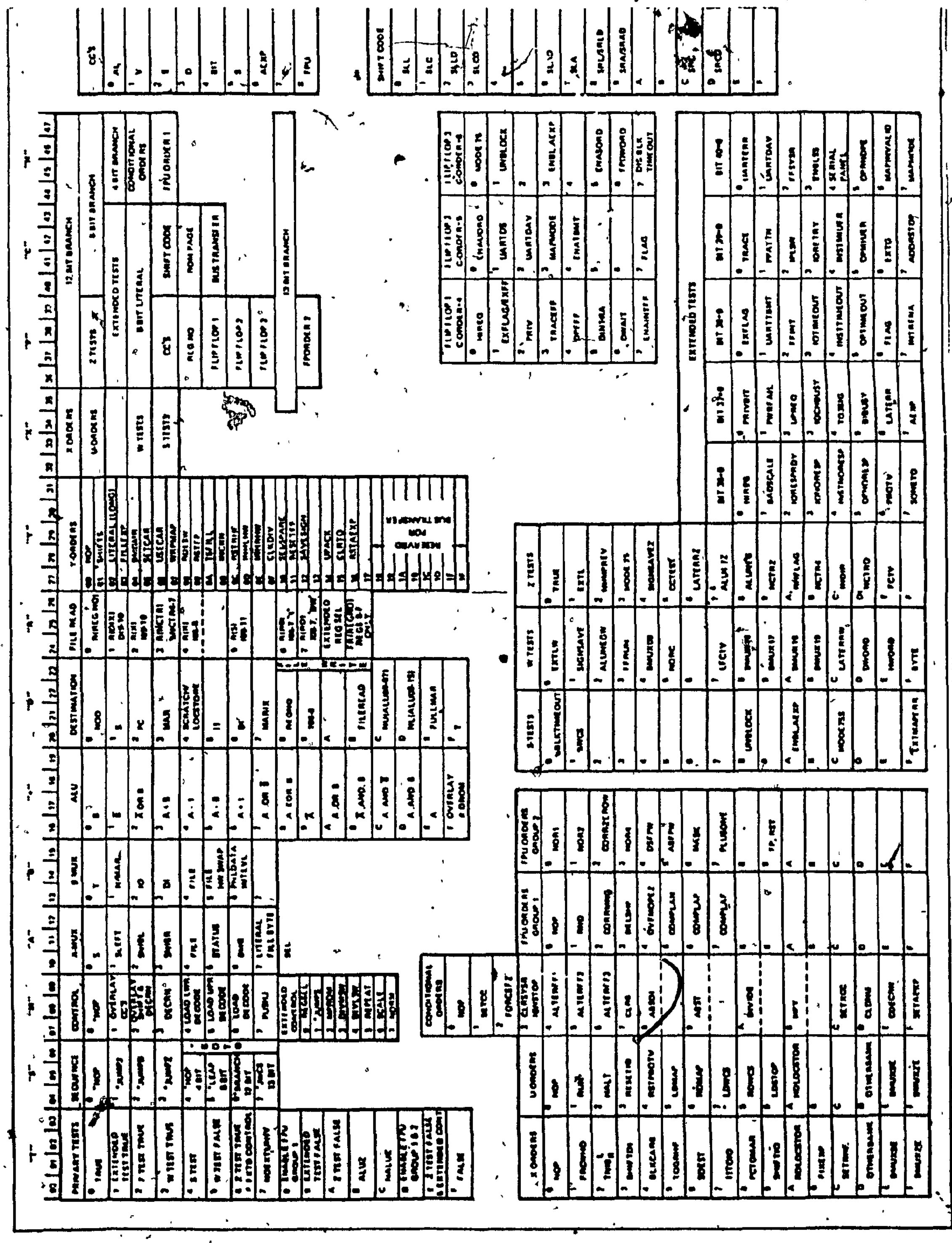


Figure 2.10 The SEL's Microword Format

in capability. She writes: "the microinstruction sequencing capabilities (should) provide the basic mechanism for implementing various forms of program logic.... Thus it is important that the microinstruction sequencing capabilities of the underlying machine organization support the implementation of the basic constructs of the appropriate program logic using context free (no embedded return address) modules of microcode. Review of the microinstruction sequencing capabilities of several contemporary microprogrammed machines has shown that these sequencing capabilities generally fail to support modular implementation of the basic constructs of flowchartable program logic" [JONE75].

Microinstructions are executed in a general fetch-decode-execute sequence, but details of actual implementation can vary greatly. Generally a microprogram counter is used to indicate the address of the next microinstruction, and a certain field may be set aside within the microword to indicate a branch address. Unlike machine language programming, the effects of the sequencing scheme are not hidden from the microprogrammer and he must cope with them.

In sequencing microinstructions there are two aspects to be considered, one is the fetch-execute cycle of the microinstructions themselves and the other is the sequencing of microoperations within each microinstruction. The first aspect covers the relationship between microoperations in different microinstructions. The second covers the relationship between microoperations within the same microinstruction.

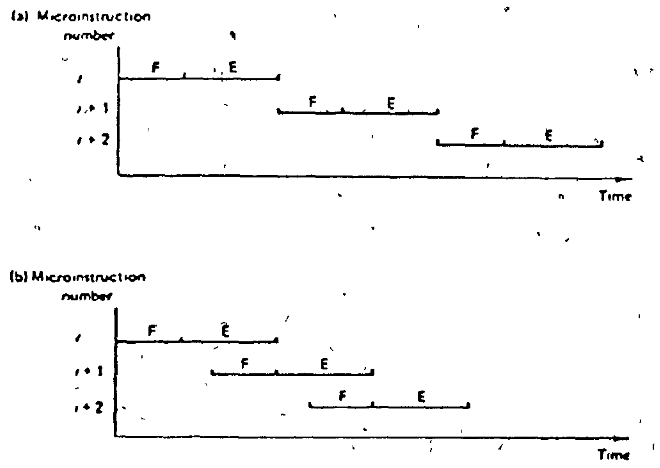


Figure 2.11 The Serial-parallel Characteristics (from RAUS80)

The first aspect is described by the serial-parallel characteristics of the sequencing scheme. In a serial implementation, fetching the next microinstruction does not begin until the execution of the current one terminates [fig. 2.11a]. In a parallel implementation, the fetch of the next microinstruction begins while the current one is being executed [fig. 2.11b]. The advantage of the serial approach is simplicity of realization, as the hardware does not have to control fetching and execution simultaneously and no special problems arise in conditional branching. The advantage of the parallel approach is the corresponding saving of time.

The second aspect of sequencing is described by the monophase-polyphase characteristics of the sequencing scheme. These refer to the number of phases (minor cycles or subcycles) used to execute a microinstruction, which usually requires one major clock cycle. In a monophase implementation there are no distinct control cycles and the microinstruction is executed by a single simultaneous issue of control signals. In a polyphase implementation each major clock cycle comprises multiple subcycles and the hardware generates control signals at each subcycle. This approach is sometimes considered a disguised form of vertical microprogramming. The advantage of monophase operation are simplicity of realization (no special hardware needed) and speed (parallel operations), however it can only be used effectively in cases where there are no data conflicts among operations within a microinstruction. The advantage of polyphase operation is that it allows interaction among resources at the expense of more complicated hardware.

The SEL 32/75 uses a microprogram counter to point to the next microinstruction. This counter can be modified by the contents of certain fields within the microword, or by the the contents of certain registers or status indicators. Otherwise it is incremented by one by a special adder. Microinstructions are executed in two cycles each of 150 nanoseconds. During the first cycle (CROM) the basic tests and sequencing are accomplished. The second cycle executes all orders that the microinstruction directs. Although the microinstruction requires the full 300 nanoseconds to execute, one microinstruction can be handled every 150 nanoseconds, the second microinstruction going through the CROM cycle at the same time the first one is being executed in the CREG cycle [fig. 2.12].

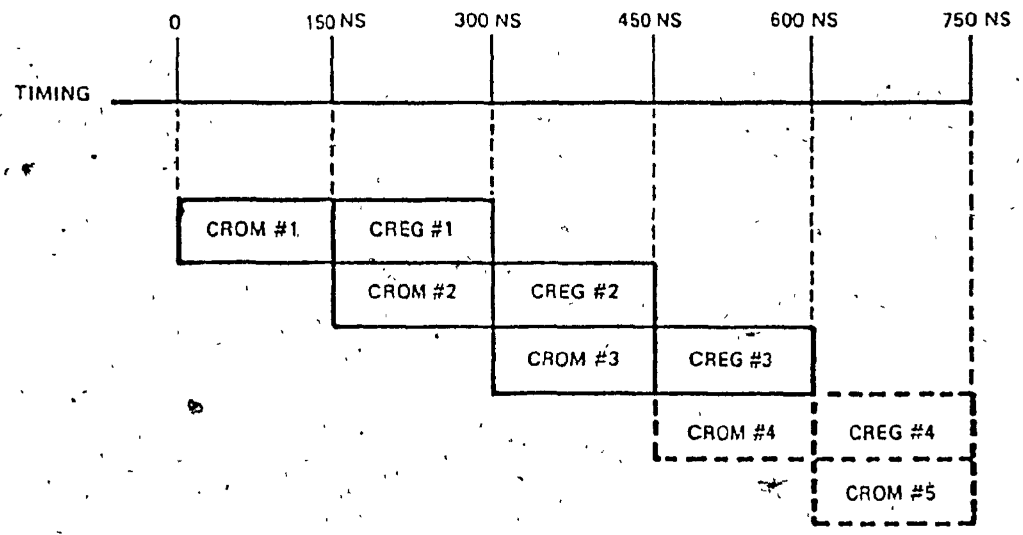


Figure 2.12 Sequencing in the SEL (from SEL1)

2.4) PROBLEMS ASSOCIATED WITH MICROPROGRAMMING

The production of microcode is usually a slow and difficult process in which familiarity with the hardware and efficiency of microcode are absolute requirements. The first is due to the level (hardware) at which microprogramming is done and the second because microprograms constitute the base of the entire software system of a computer. Chaptal writes: "microprogramming introduces an articulation between the hardware and software" [CHAP75].

Most contemporary microprogramming is done by computer manufacturers to implement the basic instruction set of their computers. In this environment, familiarity with the architecture is a natural consequence of the manufacturing process.

More often than not the microprogrammer has an electrical or electronics engineering background rather than a computer science education, with solid experience in digital logic. It is a profound knowledge of architectural components and their organization that allows the microprogrammer to make use of the parallelism inherent in the architecture, thereby producing efficient microcode.

Aids to Microprogramming	instructions per man-day	total time to produce 10000 instructions (man-years)
simulator	2	18 years
microassemblers and loaders	5-10	5-9 years
high level language	10-25	2-5 years

Figure 2.13 Microprogramming Productivity (from CHAP75)

Chaptal has compiled the table of figure 2.13 to illustrate the low productivity of microprogrammers even when working with short microinstructions (16 bits) [CHAP75]. Since long microinstructions are more desirable because of their potential for increased parallelism, it can be expected that the productivity of microprogrammers will diminish even further unless the obstacles to microprogram production are removed. These obstacles vary somewhat from machine to machine but they can be grouped into three main problem-areas: machine dependence, low level microprogramming software and microcode optimization.

2.4.1) MACHINE DEPENDENCE

The problem of machine dependency of microprograms originates from their position in the programming hierarchy of a computer, i.e., at the lowest level just above the hardware. All other programming levels are implemented on a virtual interface. High level languages produce intermediate code for a virtual machine which is then separately transformed to machine language. Even assembler language is implemented on a virtual machine, since it uses a small subset of the hardware's storage resources and in some cases even virtual structures such as stacks (ex: certain Burroughs machines). Assemblers can also invoke operations which are not directly implemented by the hardware such as MOVE LONG instructions and in the case of stack machines the PUSH and POP operations.

Microprograms, on the other hand, are based on the hardware of the particular machine. They can only reference those structures which physically exist and can invoke only those operations which actually may be executed by the hardware. Dasgupta states the problem formally as: "the fact that microprograms by definition create some desired target architecture on a machine-specific (host) microarchitecture" [DASG80].

The problem of machine dependence affects the transportability of systems implemented through microprogramming. Baba & Hagiwara write: "the desire for machine independent microprogramming systems has come from the fact that microprogramming languages are not compatible even between two adjacent models of the same architecturally compatible product line" [BABA81].

Machine dependence originates in three main areas of microarchitectural variability [DASG80]:

1) The semantics of microoperations, which refers to the various meanings that conceptually equal operations may take in different machines.

2) The timing of microinstructions, which refers to the sequencing of microinstructions as well as of the microoperations within a microinstruction.

3) The data path structure, which refers to the data transfer paths between hardware components.

2.4.2) LOW LEVEL SOFTWARE SUPPORT

The development of microprogramming languages has lagged behind that of macroprogramming languages. At a time when the construction of compilers for high level languages has become a mere routine, in the field of microprogramming this is still causing many problems [SINT80].

Most of the software available for microprogramming comes from the manufacturers in the form of microassemblers. These are very primitive in nature and are totally machine dependent. Their use requires familiarity with the microarchitecture and leaves the user to deal on his own with the timing constraints of the sequencing mechanism. In the case of the SEL 32/75, the microassembler composes one microinstruction at a time by mapping particular character strings to bit patterns in appropriate locations within the microword [SEL2]. . It provides pseudoinstructions and facilities to change the basic character strings to more meaningful ones, but no attempt whatsoever is made to provide for global composition (merging of contiguous microinstructions) or to hide the effects of the sequencing mechanism.

There have been several attempts to develop high level microprogramming languages, but they have only partially succeeded. Some, like STRUM [PATT76], are successful in producing efficient microcode but are totally machine oriented and the microprograms are not transportable. Others like SIMPL [RAMA74] were oriented towards the implementation of a particular composition technique that excluded certain control and data structures. Others like EMPL [DEWI76] tried to achieve machine independence by making the language extensible through the implementation by firmware of non-available hardware facilities, but this led to inefficient microcode. Baba & Hagiwara developed a machine independent microprogramming system [BABA81] that included a machine description and a microprogramming language, but the level of detail is so deep that it resembles a hardware description language and it is beyond the grasp of most application programmers.

2.4.3) MICROCODE OPTIMIZATION

The optimization of a microprogram is directed at reducing its execution time to a minimum and possibly its size as well. This process involves microcode compaction, i.e., the minimization of the number of microoperations, and consequently of microinstructions required to carry out a certain task. This is usually an arduous process because of the resource conflicts and data dependencies introduced by the bi-directional flow of control in microprogramming.

The problem can be stated formally as follows: "given a microprogram expressed as a sequence of microoperations, these must be placed into microinstructions so that the execution time is minimized" [LAND80]. This topic has become a major research topic because of "the almost universal belief that only optimal microprograms are useful" [DASG80] and the pressure from the increasing number of applications for microprogramming.

The problem is usually attacked by dividing the microprogram into branch free blocks and then trying to achieve local compaction in each block. Once we have a group of compacted blocks, then global optimization may be attempted. It is this last step which has not been satisfactorily solved. Shriver & Lewis write: "Landskov et al have shown that practical algorithms for the NP-hard problem of optimizing straight-line segments of microoperations are feasible, and give nearly optimal results. Indeed microcode compaction can be considered one of the few solved problems of microprogramming. Global optimization of microprograms is still a problem, however" [SHRI81].

CHAPTER 3: DESCRIBING THE MICROARCHITECTURE

We have already established the intimate relationship between microprogramming and the microarchitecture. Familiarity with the microarchitecture is essential for the production of useful microcode. So far this familiarity has remained the domain of the hardware specialist and of a few brave application programmers who have the will to expend the time and effort necessary to acquire such familiarity.

Acquiring familiarity with a computer is an arduous process. It involves, among other things, the reading of countless technical manuals filled with hardware jargon, a trial-and-error approach to microprogramming with the consequent number of system crashes, manual optimization of microprograms, and plenty of time to do all of the above, most of it wasted on trivial problems whose solution is often found in a piece of information buried deep inside one of the manuals.

The problem of familiarizing an application programmer with a given machine is then a problem of transmitting information about the machine. The information must be easily accessible and must be meaningful in terms of concepts with which the programmer is already familiar.

Making the information accessible requires that it be organized in a storage medium that allows the user access at will and with user-defined search criteria. All of the above suggest a computer-accessed database as the ideal storage medium.

Barbacci proposes a similar idea with the creation of a global database to contain machine information to be used by all types of computer applications: architecture evaluation, simulation, automatic programming etc. This machine database would be created by processing an ISPS (Instruction Set Processor Specification) description of the machine [BARB81].

Providing meaningful information requires the use of a notation with which the programmer is already familiar, i.e., concepts and constructs from the area of high level language programming. It must be noted however, that the environment of microprogramming is so distinct from that of high level programming, that there are certain structures and actions for which there are no appropriate language constructs. This situation appears in two forms:

1) Structures and actions that are so specific to microprogramming that there are no language constructs to describe them. For example, the sequencing of microoperations, serially and in parallel, can not be accurately described using the typical parallel programming constructs COBEGIN and COEND.

2) Structures and actions that may be described with known constructs, but the result is so complex as to make them less comprehensible. For example the shifting of bits in a

register may be described by a series of data transfers between the elements of an array. But then what is conceptually one operation, becomes many.

There is, therefore, a need for new language constructs especially suited for the microprogramming environment. These should, however, be kept to a minimum and should not require any major effort on the part of the programmer to understand them.

3.1) COMPUTERS AND ABSTRACT DATA TYPES

The abstract data type is a concept which permits a very clear and precise definition of data objects within a program. From the point of view of programming an abstract data type is defined as [GILO8Ø]: "a homogeneous set of data objects and a set of operations applicable on the objects of the type".

A typical example of an abstract data type is a stack. It has a well defined structure (width and depth) and associated with it some well defined access procedures which act on the data structure (PUSH, POP etc.). Heinanen et al have used this concept to describe a special-purpose computer designed specifically towards the implementation of a programming hierarchy based on some primitive abstract data types which are implemented through microprogramming [HEIN8Ø].

The microengine of a computer can be described as having a well defined data structure and a set of well defined operations which act on the data structure, however it can not be treated as an abstract data type because of the following:

i) The microarchitecture of a computer is more like a collection of abstract data types. The data structure can not be considered a single storage item but rather a collection of storage items, some of which may be connected. Each item is usually acted upon by a subset of operations and often these subsets are not disjoint, i.e., a given operation can be performed on different storage items.

ii) Abstract data types by definition provide a flexibility which is not found in the context of the microarchitecture. Among the access procedures associated with an abstract data type there are procedures to create and delete objects of the type. In the microarchitecture's environment there are no operations equivalent to such procedures. For example, if the microengine includes a counter we may describe its structure and the operations that can be performed on it, but we can not delete it or create additional counters. We can only use whatever counters physically exist in the microengine.

While the concept of an abstract data type can not be applied strictly in the description of the microarchitecture of a computer, it can be used as a guide to provide a description in the form of hardware abstractions of the components of the microengine.

3.2) THE NOTATION: A HARDWARE ABSTRACTION LANGUAGE

The description of any structure or process requires the existence of a coherent notation that is understandable by those to whom we wish to communicate the description. In order to describe a microprogrammable computer, the notation must meet the the following requirements:

1) The machine must be described at the microarchitecture level. There are several levels of abstraction in a machine. These range from the purely functional level of the macroachitecture down to the level of the individual gates. The notation must therefore be tailored to the required level of abstraction.

2) The use of the notation must yield a functional description of the microarchitecture. The structure of storage components must be described with a few basic types. The semantics of operations must be described using a few well defined primitive operators. Both the basic types and primitive operators must be easily understood by the prospective microprogrammer.

To meet the above requirements this thesis proposes a Hardware Abstraction Language (HAL) for the functional description at the microarchitecture level, of the structures and operations physically occurring in the hardware.

The purpose of HAL is to describe the microarchitecture to the prospective microprogrammer in the form of a set of well defined storage structures and a list of well defined operations on them. HAL is intended to be applicable to description of a wide range of microarchitectures. However, its design has been influenced by its application to the SEL32/75 during the course of this research.

HAL is not an executable language, but a hardware description language designed to yield a functional description of the microarchitecture. This last point must be emphasized, since the syntax of HAL is flexible enough to allow the specifications of structures and operations for which a process of code generation would seem very complex. One must keep in mind, however, that the processing of a HAL description will not generate code, but a machine information file. This file would be used by a microcode generator, and by the prospective microprogrammer to get acquainted with the microarchitecture. It could possibly be used by a microprogram verification (simulation) system.

Regardless of the complexity of the structures and operations described using HAL, one must remember that whatever is described must already be implemented in the hardware. The objective of HAL is to describe what physically occurs in the hardware, and not to implement new structures and operations based on existing hardware items, as is the case with most high level languages.

3.2.1) THE JUSTIFICATION FOR HAL

The first and most important justification for HAL is the need for a description of the microarchitecture. The problem then is to justify the choice of a particular notation to satisfy this need.

In choosing a particular notation we must take into account the following considerations:

1) The guiding strategy in describing the microarchitecture is the concept of hardware abstraction, i.e., the microengine is to be described as having a set of well defined storage structures with a set of well defined operations on them.
2) The purpose of the microengine description is to provide information to the microprogrammer as well as to the microcode generator.

Upon considering the choice of a notation to describe the microarchitecture, one is inclined to use one of the existing hardware description languages. These have, however, been found to be unsuitable due to the following reasons:

1) Computers are described at several levels, none of which can accommodate exactly the description of the microengine. Baer describes the following levels [Baer80]: The global system (PMS), the processor description (ISP), the register transfer level, and the logical design and circuit level. The processor description level encompasses the microengine description, but notations such as ISP are concerned with providing information for the assembler programmer. Bell & Newell state: "The ISP descriptive system is meant to provide a uniform way of describing instruction sets, that is, of giving the information contained in a programmers manual" [BELL71]. The instruction set of a computer is however built upon the storage structures and functional capabilities of the microengine, and these are more numerous and varied than what the assembler programmer perceives.

2) Hardware description languages are not specifically concerned with microprogramming and their scope goes beyond the microengine. This usually creates an unnecessary degree of detail. The biggest defect of these languages is that they do not establish a link between the functional description of operations and the microinstructions that implement those operations. Dembinski & Budkowski state: "the specific feature of

microprogramming is the direct link between a functional specification of hardware units and microprogram instructions. A microprogram designer must be aware not only of how to express his algorithmic solution of a particular problem, but he also has to [take] care of the hardware realization of his algorithm" [DEMB78].

3) The level of detail shown through hardware description languages is usually beyond the grasp of most application programmers. Even in cases where the language is concerned with microprogramming as in the MPG system [BABA81], the objective is to provide information for the microprogram compiler and not for the prospective microprogrammer.

4) The sequencing of microoperations can not be adequately described by the timing primitives of existing languages. Sint writes [SINT81]: 'In most languages, one can only distinguish sequential and parallel execution of operations. For a description language that ..., describes microoperations as indivisible units this is insufficient, since their execution can overlap without being fully parallel'.

Given the unsuitability of existing hardware notation, HAL was designed with the following objectives in mind:

1) To describe the storage structures and functional capabilities of the microarchitecture in a form which application programmers could understand with minimal

effort and at the same time be able to provide information to the microprogram generator.

2) To establish a direct link between the functional description of operations and the microinstructions that implement those operations.

3) To be able to create a machine information file by processing the HAL description of the microarchitecture. This file is to be used both by the microprogrammer to get acquainted with the microarchitecture and by the microprogram generator to transform a microprogram specification into microcode.

3.2.2) THE SYNTAX OF HAL

Freeman describes modern computers as having three types of circuits: "storage, data transfer and manipulation, and control. Storage circuits are able to hold information over time. Data transfer and manipulation are the guts of a computer and provide the information processing power of a computer. The essential third ingredient of a modern stored-program computer, the control circuitry, provides the ability to evoke the operations of other circuits in many different sequences (i.e., different programs)." [FREE75]. It is along the lines proposed by Freeman that HAL describes the microarchitecture. Since our purpose is to describe the physical circuitry through abstractions, we shall refer to Freeman's three types of circuits as: the data structure, the functional operations, and the control structure.

HAL is a language used to produce an abstraction of a machine by listing its storage items, its functional operations and its control operations. It is similar to Pascal in the description of the data structure and in the basic functional statements. HAL was designed with the description of the SEL 32/75 in mind, but it is flexible enough to accommodate other machines and it could be easily extended. In BNF we write:

```
<machine> ::= MACHINE <identifier>;
              <machine description>
              END.

<machine description> ::= <data structure>
                          <functional operations>
                          <control structure>
```

The data structure includes the description of the data types used as well as of the storage items available to the microprogrammer. The abstractions of existing storage items is done by using certain basic types to build more complex types. In BNF we have:

```
<data structure> ::= <data types> <storage>

<data types> ::= TYPE <types> !
                 <empty>

<types> ::= <type declaration>; !
            <type declaration>; <types>

<type declaration> ::= <identifier>= <type>

<type> ::= STACK[<positive constant>] OF <type> !
           SEQ (<constant>..<constant>) OF <bit type> !
           ARRAY [<index type>] OF <type> !
           TUPLE <field list> END !
           <simple type>

<bit type> ::= BIT ! ZERO ! ONE
```

```
<index type> ::= <simple type> !
                 <simple type>,<index type>

<field list> ::= <identifiers declaration> !
                 <identifiers declaration>;<field list>

<identifiers declaration> ::= <identifier list>: <type>

<identifier list> ::= <identifier> !
                      <identifier>,<identifier list>

<simple type> ::= <identifier> !
                  <bit type> !
                  <constant>..<constant>
```

The basic types used by HAL are BIT, ZERO, and ONE. These are defined as:

```
TYPE BIT= Ø..1;
     ZERO= Ø..Ø;
     ONE= 1..1;
```

These basic types accept the usual arithmetic operations +,-,*,/ as well as the boolean operations OR,AND,XOR,NOT. A value of 1 stands for true and a value of Ø stands for false. Types ZERO and ONE are required because storage items sometimes have certain fields permanently set to 1 or Ø. An item of type ZERO is always false and one of type ONE is always true.

The type SEQ is used to describe strings or sequences of bits. It is similar to the packed array of 1 dimension in Pascal, but with the difference that one can operate on individual bits as well as on the whole item. Bit labelling is established through a range of indices denoted by constants. The range of bit indices need not go from low to high, it may go from high to low if it reflects the manufacturer's convention. The most typical example of a sequence structure is a register. Consider for example the following description of an X register in the CYBER:

```
TYPE XREG= SEQ (59..0) OF BIT;
```

in this case we chose 59..0 because it corresponds to CDC's convention of bit labelling. Whatever the choice, one must be consistent in order to have clarity. Allowing the use of high-low as well as low-high ranges seems to complicate implementation, but one must remember that HAL is not an executable language but a descriptive language from which no executable code will be generated.

The type STACK refers to the usual last-in-first-out structure with its associated PUSH and POP operations. In some machines such items are part of the hardware. The SEL32/75, for example, has a hardware stack which it uses to store return addresses during subprogram processing. The type STACK is not meant to describe stacks which may be perceived at the assembler level, but which do not exist in the hardware. The positive constant associated with the type STACK indicates the depth of the stack. For example in the SEL there is the JSTACK which can be described as follows:

```
STORE JSTACK: STACK[4] OF SEQ (0..12) OF BIT;
```

The type TUPLE is a structure which consists of a number of components or fields. It is similar to the Pascal RECORD, except that it has no variant part. An example would be the description of a floating point register type:

```
TYPE FPREG= TUPLE
             SIGN: BIT;
             EXPONENT: SEQ (1..11) OF BIT;
             COEFFICIENT: SEQ (12..59) OF BIT
            END;
```

If there is a register X of type FPREG then one can refer to one of its components, say EXPONENT, by coding X.EXPONENT.

The storage part of the data structure describes the storage items available in the microarchitecture. Every item described in this section must be based on the hardware, either the whole item or in case of a TUPLE its components. This allows for the logical grouping in a TUPLE of items which exist in the hardware but which are not physically grouped. In BNF the storage description is defined as follows:

```
<storage> ::= STORE <storage items>

<storage items> ::=
 <identifiers declaration>; !
 <identifiers declaration>; <storage items>
```

An example of storage declaration could be:

```
TYPE REG: SEQ (31..0) OF BIT;

STORE R0,R1,R2,R3: REG;

      STATUS: TUPLE
               ALUPOS,ALUZERO,OVFLOW: BIT;
               PROGCTR: SEQ (12..0) OF BIT;
               KEY: SEQ (2..0) OF BIT
```

```
        END;
```

The item STATUS may not necessarily exist as a whole in the hardware, but one may group several separate hardware items to establish a logical grouping.

The functional operations refer to the capabilities of the machine in data transfer and manipulation. The objectives of this section are:

1) To establish the set of operations which the microprogrammer may invoke in the specification of a microprogram.
2) To establish the possible consequences of each operation.
3) To establish a direct link between the abstract description of the operations and the actual microword that carries out those operations.

The functional capabilities of the machine are described as a list of operations. In BNF we have:

```
<functional operations> ::=  OPERATION <operation list>

<operation list> ::= <operation> !
                   <operation>; <operation list>

<operation> ::=
 EXP: <explicit operations> <underlying operations>

<explicit operations> ::= <statement> !
                      <compound statement>

<compound statement> ::= COCYCLE <statement list> END

<statement list> ::=
 <statement> !
 <statement><sequence operator> <statement list>
```

```
<sequence operator> ::= , ! ;
```

Explicit operations are those that can be invoked directly by the microprogrammer. The operations are decribed using statements which correspond to abstract descriptions of the physical actions occurring in the hardware. The compound statement is used to describe the occurrence of multiple operations within a single major clock cycle. It is understood that each microinstruction takes one major clock cycle to be processed. The originator of the COCYCLE construct is S. Dasgupta (see DASG80). It is used here because it best describes the parallelism of the microengine. Sequence operators are provided to indicate serial (;) or parallel (,) realization of operations. In a polyphasic machine, actions taken during different subcycles are separated by semicolons. Actions occurring during the same cycle or subcycle are separated by commas.

Underlying operations describe the operations that occur as a consequence of the explicit operations invoked. They also describe the microinstruction used to realize the explicit operations. In BNF we have:

```
<underlying operations> ::=
 IMP: <implicit operations> MIW: <microword> !
 MIW: <microword>

<implicit operations> ::=
 <implicit operation> !
 <implicit operation>; <implicit operations>

<implicit operation> ::= <statement> !
                        <compound statement>
```

Implicit operations usually describe data transfers or storage settings which result from the execution of the explicit operations, but which can not be directly invoked by the microprogrammer. The setting of condition codes after an arithmetic operation is a typical example of implicit operations. Implicit operations may be specified as a list of sequential operations because some explicit operations may take more than one major clock cycle to complete. For example in the SEL32/75 writing to a general register is initiated in one cycle and completed in the next.

Explicit operations are linked to a microinstruction by the description of a microword in the form of a string of bits. Each bit is described as having a fixed value (Ø,1), a variable value (letter), or as not being used (-). In BNF we have:

```
<microword> ::= '<bit string>'

<bit string> ::= <bit description> !
                <bit description><bit string>

<bit description> ::= Ø ! 1 ! - ! <letter>
```

The microword description must be such that if processed by the microengine the storage items involved would be affected in the way described by the explicit and implicit operations.

Statements in HAL are Pascal-like except for the introduction of concatenation operators and some new keywords. In BNF we have:

```
<statement> ::= <simple statement> !
               <conditional statement>

<simple statement> ::=
 START <variable concatenation>:= <expression> !
 <variable concatenation>:= <expression> !
 NULL:= <expression> !
 <predefined procedure call>

<conditional statement> ::= ON <expression> DO (<actions>)

<actions> ::= <simple statement> !
            <simple statement>,<actions>
```

The keyword START is used to indicate that there is not a definite completion time associated with this operation, i.e., the microprogrammer may invoke the operation but he can not expect it to complete within a fixed predetermined time. Operations which are completely internal to the microengine always complete in fixed time, however in cases when the microengine has to communicate with external devices, the time elapsed until completion may vary depending on how busy the external device is. An example of this situation occurs during main memory access in which the elapsed time varies according to the traffic in the memory bus.

The keyword NULL indicates that the result of the expression is not to be transmitted anywhere. Usually the purpose of such operations is to test the value of the expression, the result of the test being determined by the implicit operations.

Predefined procedures will be used to describe certain operations which are specific to the microengine and which are not easily expressed in high-level notation. These will be described in detail later.

The semantics of conditional statements are that if the expression is evaluated as true, then the operations specified as actions will be carried out. Conditional statements allow conditional interaction among resources in a polyphasic machine.

Variable concatenation describes the grouping of storage items for the purpose of receiving or providing information. The operations are similar to those of string concatenation in string-oriented languages. In BNF we have:

```
<variable concatenation> ::=
 <variable reference> !
 <variable reference>&<variable concatenation>

<variable reference> ::= <variable> !
                        <variable>(<bit index>)

<bit index> ::= <expression> !
               <expression>..<expression>

<variable> ::= <identifier><specifiers>

<specifiers> ::= <specifier><specifiers> !
               <empty>

<specifier> ::= .<identifier> !
               [<expression list>]

<expression list> ::= <expression> !
                     <expression>,<expression list>
```

The bit index allows partial access to a storage item, i.e., to some of its bits only, while the specifiers allow access to specific components of a structured item, both ARRAY and TUPLE types.

Expressions in HAL are similar to Pascal except for the inclusion of some new operators, such as XOR which are commonly available at the microprogramming level. In BNF we have:

```
<expression> ::=
 <simple expression> !
 <simple expression> <relational op> <simple expression>

<relational op> ::= # ! = ! < ! > ! <= ! >=

<simple expression> ::=
 <sign><term> !
 <sign><term> <add op> <simpler expression>

<simpler expression> ::=
 <term> !
 <term> <add op> <simpler expression>

<sign> ::= + ! -

<add op> ::= OR ! XOR ! + ! -

<term> ::= <factor> !
           <factor> <multiply op> <term>

<multiply op> ::= AND ! * ! /

<factor> ::= <concatenation> !
             <predefined function reference> !
             (<expression>) !
             NOT <factor>
```

Predefined functions serve the same purpose as predefined procedures and will be discussed later. Concatenations are similar to variable concatenations except that they allow the introduction of constants or literals in the concatenated item. In BNF we have:

```
<concatenation> ::= <element> !
```

```
                    <element>&<concatenation>

<element> ::= <positive constant> !
              <variable reference> !
              <literal indicator>

<literal indicator> ::= $<letter>
```

Literal indicators are used to indicate the position within a concatenation of a modifiable literal. These allow the introduction of constants by the microprogrammer in an operation. As an example, let us consider the description of register operations in a machine. Since registers usually have the same characteristics, it is not unusual to treat them as an array. Consider a machine with 8 general registers (Ø-7) and a shift register. They could be described as follows:

```
STORE GREG: ARRAY [Ø..7] OF SEQ (31..Ø) OF BIT;
      SREG: SEQ (31..Ø) OF BIT;
```

A typical operation in such a machine would be the transfer of data from a general register to the shift register. Using a literal indicator, we could describe a general operation as:

```
EXP: SREG:= GREG[$R]
IMP: .............
MIW: '..........RRR...........';
```

The letter R would occupy the field within the microword which determines the bounds of the value which can be used. When the microprogrammer codes a microprogram he would then specify the the particular register by coding:

```
SREG:= GREG[$R6]
```

The microcode generator updates the microword with the value required, 6 in this case.

The control structure section is used to describe the timing of tests and the control operations which can be performed in the machine. In BNF we have:

```
<control structure> ::=
 <tests description> <control operations>

<tests description> ::= TEST <test list>

<test list> ::= <test>; !
             <test>; <test list>

<test> ::= <expression> <timing operator> <time>

<timing operator> ::= AT ! AFTER

<time> ::= <sign> <phase>

<phase> ::= <positive integer> !
         <positive integer>.<phase>
```

The tests description establishes the testable conditions in the machine as described by an expression, and the timing constraints on the testing of those conditions. The keyword AT indicates that the condition is available for testing only during the given cycle or subcycle, and not before or after. The keyword AFTER indicates that the condition is available for testing, and remains so, after the given cycle has expired. These timing constraints usually originate in the scheme used to sequence microinstructions.

An example would help to illustrate the description of testable conditions. It is understood that conditions are set during the current clock cycle, i.e., cycle 0. Consider two storage items ALUZ and COUNTZ. The first one is set to 1 if the result of an ALU operation is zero, otherwise it is set to zero. COUNTZ is set to 1 if the value of a counter becomes zero, otherwise it is set to zero. Assume there is no pipelining of microinstructions, i.e., there is no overlapping of the cycle times of contiguous microinstructions. Under these timing constraints, a description of these tests is as follows:

```
TEST ALUZ = 1 AT +1;
     COUNTZ = 1 AFTER 0;
```

The use of the keyword AT with ALUZ indicates that the ALU's bits can only be tested in the cycle inmediately following that in which they were set. In both of the above cases one cycle (cycle 0) has to elapse before the condition can be tested, but the first condition can be tested only during the second cycle, while the second condition can be tested anytime beginning with the second cycle.

The control operations section describes the ability of the machine to perform condition dependent non-sequential processing. The purpose of control operations is to be able to choose an address other than the next sequential address from which to fetch the next microinstruction. The operations are described using very primitive control constructs which express the different types of program flow modification. In BNF we have:

```
<control operations> ::= CONTROL <control list>
```

```
<control list> ::= <control operation>; !
                   <control operation>; <control list>

<control operation> ::=
 EXP: <control description> <underlying operations>

<control description> ::=
 <condition> <control operator> <concatenation>

<condition> ::= ON <expression> !
               <empty>

<control operator> ::= GOTO ! GOSUB ! RETURN
```

The control operators determine the kind of branching that may occur. They all have the ability to interrupt sequential processing but each has different implications. The operator GOTO achieves a simple branch at the same process level. The operator GOSUB is used to branch down to a lower process level, i e, a subprogram, implying that provisions are made to eventually return to the same level. These provisions usually include saving the state of the machine for future restoration. The RETURN operator is used to branch up to a higher process level, implying a restoration of the original state of the machine. The implications of any of these operators will be described in the underlying operations.

The concatenation in the control operation provides the source of the branch address. This may be indicated by a constant, a storage item, a literal indicator, or a combination of them. The specification of a storage item indicate that its contents provide the branch address.

The predefined procedures and functions mentioned before are used to describe operations which are very specific to the microarchitecture and can not be easily described in terms of simpler operations. Different machines may include such operations and the notation (HAL) can be extended by considering them as primitive operations. Typical of these operations are shift operations on registers, push and pop operations on stacks, and a 'select' operation to perform partial transfers of the result of an expression. These are described in detail below.

Shift operations on registers may be described as a series of steps moving the bits of a sequence in a certain direction a certain number of times. Such a description, however, would make the shift operation less comprehensible. It is therefore better to consider the shift operation as a primitive and to define clearly its semantics.

Shift operations may appear in either procedure or function form. It is understood that functions have a value associated with them and are used within expressions, while procedures do not and may be used independently [WIRT73]. In both forms the information required is the same: the direction of the shift (right, left), the amount of shift, the fill bits, and the item to be shifted.

In procedure form, the shift operation will be designated by SHIFTC to indicate a shift of the content of the storage item involved. In this form the storage item is affected permanently and may acquire a different value. In function form the the shift operation will be designated by SHIFTV to indicate a shift of the value of the storage item involved. In this form the contents of the shifted item itself is not affected.

PUSH and POP are the usual stack operations. PUSH effects a transfer from a storage item onto a stack, and POP transfers a data item from a stack into a storage item. The SELECT operation is used to choose a string of bits from the result of an expression. For example the ALU of a machine always handles the full word width, but sometimes one can load through the ALU an item whose width is less than a word. In this case we must use the SELECT function to describe which bits of the ALU's output are to be transferred.

The syntax for these predefined procedures and functions is as follows:

```
<predefined procedure call> ::=
 SHIFTC(<shift parameters>) !
 PUSH(<storage id>,<stack id>) !
 POP(<stack id>,<storage id>)

<stack id> ::= <variable concatenation>

<storage id> ::= <variable concatenation>

<predefined function reference> ::=
 SHIFTV(<shift parameters>) !
 SELECT(<bit index>,<expression>)

<shift parameters> ::=
 <direction>,<quantity>,<fill bits>,<shift item>
```

```
<direction> ::= RIGHT ! LEFT

<quantity> ::= <concatenation>

<fill bits> ::= <concatenation>

<shift item> ::= <variable concatenation>
```

The syntax of the shift operations is flexible enough to describe any kind of shifts, as shown by the following examples:

```
EXP: SHIFTC(RIGHT,IR(9..12),Ø,GREG[$R]
IMP: .........................
MIW: '.........RRR..............';
```

The above example describes the shifting of a general register to the right by a quantity indicated by bits 9-12 of the instruction register, using Ø as the fill bit. This is a logical shift since the fill bit is zero. If the fill bit were the high order bit then it would be an arithmetic shift, while if it were the low order bit then it would be a circular shift.

```
EXP: SHIFTC(RIGHT,1,MDR(Ø),MAR&MDR)
IMP: ..........................
MIW: '................................';
```

The above example describes a double-right-circular shift of the memory address register concatenated with the memory data register. The shift quantity is 1 and the fill bit is the low order bit of the MDR.

The definition of the syntax of HAL would not be complete without the definition of its most basic elements, such as identifiers and constants. In BNF we have:

```
<identifier> ::= <letter><characters>

<characters> ::= <character><characters> !
                 <empty>

<character> ::= <letter> ! <digit>

<constant> ::= <sign><positive constant>

<positive constant> ::= <decimal> !
                        X'<hex number>' !
                        O'<octal number>' !
                        B'<binary number>'

<decimal> ::= <digit> !
              <digit><decimal>

<digit> ::=
 Ø ! 1 ! 2 ! 3 ! 4 ! 5 ! 6 ! 7 ! 8 ! 9
```

The letters referred to above are the letters of the English alphabet. Numeric constants are considered a convenient representation of binary numbers. The complete grammar of HAL is listed in appendix B.

The syntax of HAL is flexible enough as to allow a great variety of storage structures and operations. However this does not mean that they actually occur in a given machine. In fact most structures and operations found in real machines are rather simple compared with the kind that HAL could describe. The objective of HAL is to facilitate a meaningful description of structures and operations as they are observed to occur in a given machine. A description is meaningful if (1) it is able to communicate architectural information to the prospective microprogrammer, and (2) at the same time can provide information to the microcode generation process.

HAL is meant to be used by a qualified person to describe a real machine. Familiarity with the microarchitecture, the operations, the sequencing mechanism, and the microword format are absolute requirements to produce a meaningful machine description. Someone from the manufacturing environment is ideally suited for this purpose, although at a given installation someone may be designated to get acquainted with the microarchitecture and then produce its HAL description. Prospective microprogrammers would then refer to this description in order to microprogram.

CHAPTER 4: SAMPLE MACHINE DESCRIPTIONS

In this chapter I shall illustrate the application of HAL by describing various items in two sample microarchitectures. Given the flexibility of HAL, there may exist a question of style, that is two different persons may produce different abstractions of the same hardware item. In order to minimize the effects of style we must stay as close as possible to the idea behind the design of a particular hardware item. Usually this information is provided by the physical structure and the uses of the item.

The sample microarchitectures to be described are the SEL32/75 and the AM2900. The emphasis will be on the SEL32/75 since it will be used for practical experiences. The AM2900 is discussed briefly due to its importance in architecture research.

4.1) A DESCRIPTION OF THE SEL32/75

The data structure of the SEL 32/75 is shown in figure 4.1. In the following pages I shall describe the most relevant items and produce abstractions for each. The information on each item is provided by the manufacturer in the Writable Control Storage (WCS) User's Manual.

The ALU is a two-input, 32-bit Arithmetic and Logical Function unit. It can generate 15 Arithmetic and Logical Functions for two inputs. These functions are shown in the

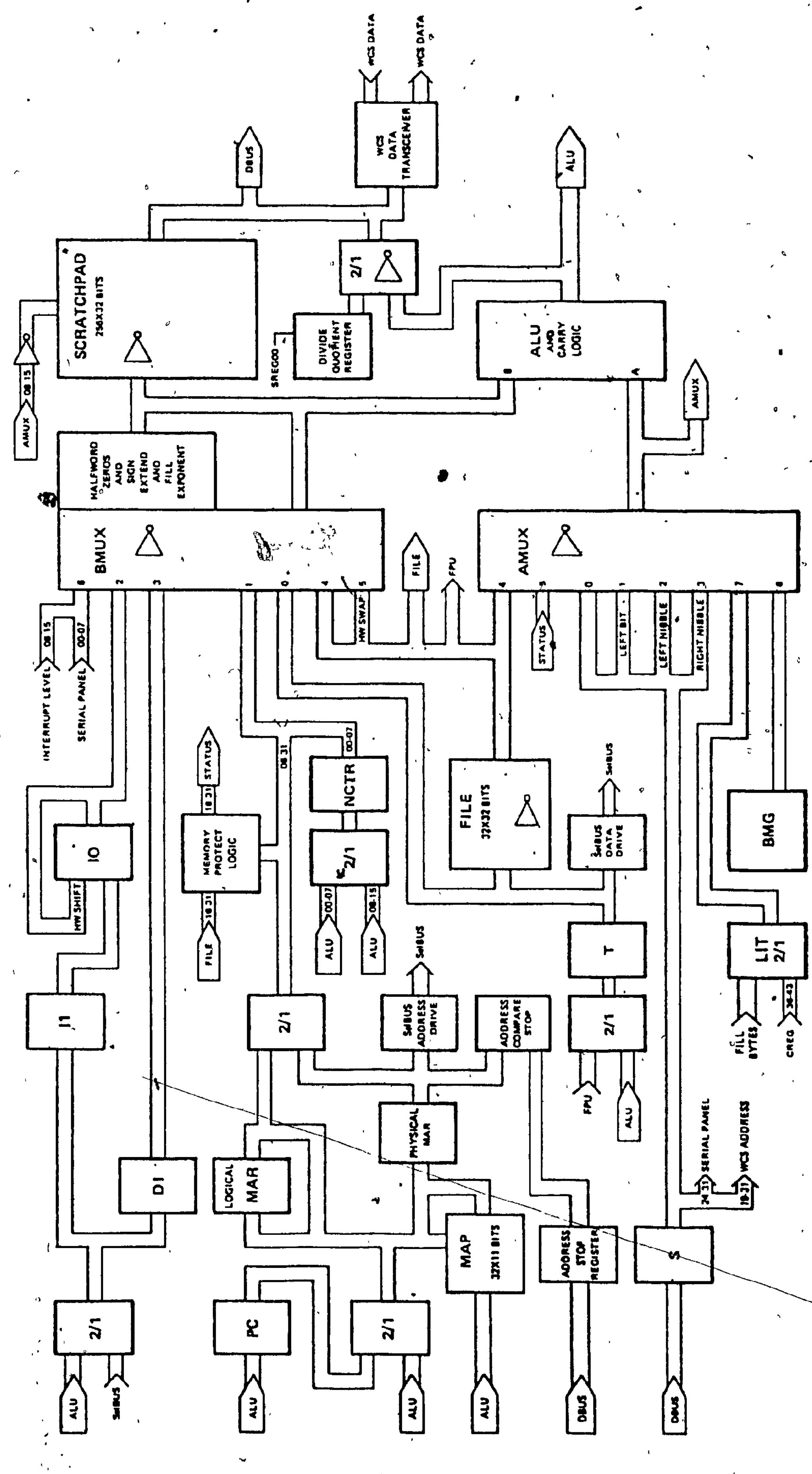


Figure 4.1 The SEL's Data Structure

'+' field of the microword format (fig. 2.10) and they determine the arithmetic and logical operators used in describing operations in the SEL. The inputs to the ALU are selected by two multiplexers (A-mux, B-mux). The output of the ALU may or may not be specified. If not specified the ALU results are used for testing purposes only.

The Literal generator (LIT) forms an 8-bit constant for insertion in one of the four bytes composing the 32-bit word. The position of the constant and the fill bits for the three remaining bytes is determined by the microcode.

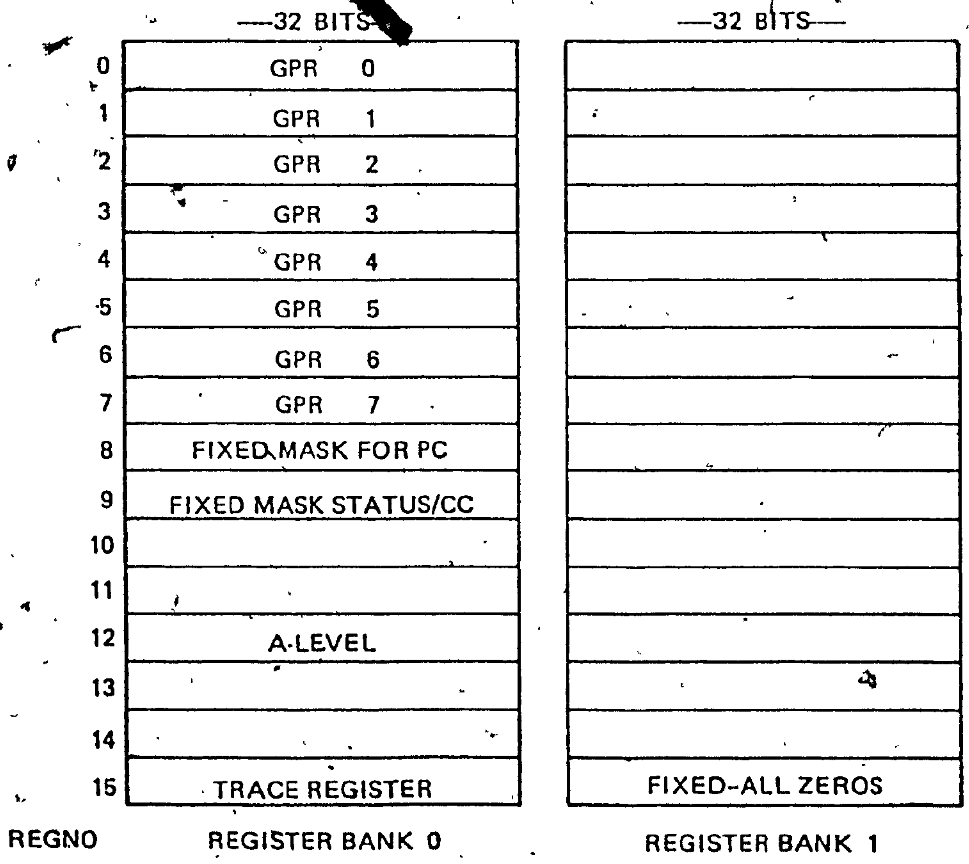


Figure 4.2 The Register File

The General Register File is a 32x32-bit multiport file memory. Each register in the file is directly addressable for read or write operations. The file is organized into two banks of 16 registers each as shown in figure 4.2.

The Memory Address Register (MAR) is a 24-bit register used to hold the address for addressing Main Memory or an I/O processor. The MAR may also be used as a temporary register and when coupled with the N-counter (NCTR, 8 bits), will form a 32-bit temporary register.

The Program Counter register (PC) is a 22-bit binary counter containing an abbreviated address of the most recent instruction fetch from memory. The PC supplies bits 8-29 to the MAR for instruction fetches. The PC is incremented automatically by the hardware when an instruction is fetched.

The N-counter register (NCTR) is an 8-bit binary up/down counter used as an iteration counter for repetitive operations such as shift, multiply, divide, etc.

The Shift Register (S) is a 32-bit temporary register which can be used for temporary storage or as a shift register. The content of the S register can be shifted in place, or may be made available in shifted form through the A-mux. This last operation does not affect the content of the S register itself. The S register can be coupled with th T or DI registers for doubleword shifts. In these cases the S register always contains the most significant part of the word. The types of shift available are: shift 1 bit, right or left, in an arithmetic, logical or circular fashion and shift 4 bits (nibble) right or left.

The Temporary register (T) is 32-bit multipurpose register used to hold data to be written to the General Register File or transmitted to main memory or to an I/O processor. It also functions as a shift register for right or left nibble (four bits) shifts.

The data input (DI) register is a 32 bit multipurpose register used to receive operands from Main Memory or data and status from I/O processors. It can also be used for bit shifts (right, left, arithmetic, logical, circular) and may be coupled with the S register for doubleword shifts.

The Instruction Decode Register (IØ) is a 32-bit register which contains the current machine instruction being executed. The IØ register is able to perform halfword shifts to accommodate halfword machine instructions.

The Instruction Pipeline Register (I1) is 32-bit register which receives machine instructions from main memory. The I1 register contains the next machine instructions to be executed.

The Local Store (SCRATCHPAD) is a 256x32-bit RAM storage array for fast access data storage. It usually contains operating system tables and information.

Given the above information, a description of the data structure could be as follows. We would first establish the basic types:

```
MACHINE SEL3275;

 TYPE WORD= SEQ(Ø..31) OF BIT;
      MWORD= SEQ(Ø..47) OF BIT;
      REGISTER= SEQ(Ø..31) OF BIT;
      REGBANK= ARRAY[Ø..15] OF REGISTER;
```

the type WORD describes the basic computer word used on most data paths throughout the microengine. The type MWORD describes the microword used for the control storage. The type REGBANK describes a bank of registers used in the general registers file. Once we have established the basic types we can then proceed to describe the storage resources in the SEL as follows:

```
 STORE S, T, IØ, I1, DI: REGISTER;
       LIT, NCTR: SEQ (Ø..7) OF BIT;
       PC: SEQ (Ø..21) OF BIT;
       MPC: SEQ

       SCRATCH: ARRAY [Ø..255] OF WORD;

       REGFILE: ARRAY [Ø..1] OF REGBANK;

       MAIN: ARRAY [Ø..262143] OF WORD;

       MAR: SEQ (Ø..23) OF BIT;

       CSTORAGE: TUPLE
                    CROM: ARRAY [Ø..4095] OF MWORD;
                    WCS: ARRAY [4096..6143] OF MWORD
                  END;

       JSTACK: STACK [4] OF SEQ (Ø..12) OF BIT;
```

The MPC declared above is the microprogram counter used for sequencing microinstructions. It is a rather complex hardware item which uses a multiplexer to choose from among several sources the next address, but it can be treated as a 13-bit register for all practical purposes. The JSTACK is a hardware stack used for storing return addresses during microroroutine calls. Its depth limits the level of nesting of microroutines. MAIN describes the main memory (Mp) available in the SEL 32/75. CSTORAGE describes the control memory used to hold microprograms. It is represented as a tuple containing two contiguous arrays. The first one is the Control Read Only Memory containing microcode provided by the manufacturer, which is available for read and execute operations only. The second array is the Writable Control Storage used to hold user microprograms. It is available for read, write and execute operations.

Besides the above structured storage items the SEL offers a variety of scalar storage items used mostly for testing purposes. They are flip-flops which can be set directly or indirectly by the microprogrammer through appropriate microinstructions. I shall describe below some of the most relevant:

```
ALUZ, NALUZ, ALUNEG, ALU47Z: BIT;
ALUSIGN: BIT;
BIBUSY: BIT;
HIREG: BIT;
```

ALUZ, NALUZ, and ALUNEG are set to 1 if the result of an ALU operation was zero, non-zero and negative respectively.

ALU47Z is set to 1 if bits 4 and 7 of the result of an ALU operation were zero. ALUSIGN is used to save the value of the sign bit (bit Ø) of the result during an ALU operation. ALUSIGN can be set directly by the microprogrammer in contrast to the other ALU bits which are set automatically by the hardware during ALU operations. BIBUSY is used to indicate outstanding transactions in the I/O bus. It is set to 1 automatically by the hardware when a bus transaction is initiated, such as a memory read, and set to Ø by the bus interface upon completion of the transaction. HIREG is used to select one of the two register banks available in the general register file. A value of Ø selects the lower bank and a value of 1 selects the upper bank. HIREG can be set by the microprogrammer at will either permanently or for the duration of 1 major clock cycle.

In the above declarations, we established the structure of several storage items in the SEL, but there was no indication as to the kinds of operations which can be performed on those items. I shall now proceed to describe various operations involving the items described above. Some items are involved in functional operations while others are involved in control operations. The functional operations follow first:

```
OPERATION

 EXP: COCYCLE
      DI:= REGFILE[HIREG][$R],
      ALUSIGN:= REGFILE[HIREG][$R](Ø)
```

```
        END
    IMP: COCYCLE
     ON REGFILE[HIREG][$R] = Ø DO (ALUZ:=1,NALUZ:=Ø,ALUNEG:=Ø),
     ON REGFILE[HIREG][$R] < Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=1),
     ON REGFILE[HIREG][$R] > Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=Ø)
        END
    MIW: '-------------1ØØØØØØØ11ØØØØ1ØØ1Ø----RRRR---------';
```

The above operation describes a transfer of a register in the bank indicated by HIREG to the DI register, simultaneously saving the sign bit. The ALU bits are set as indicated by the implicit operations. The microword described if processed by the microengine will achieve the desired results.

```
    EXP: NULL:= S
    IMP: COCYCLE
          ON S = Ø DO (ALUZ:=1,NALUZ:=Ø,ALUNEG:=Ø),
          ON S < Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=1),
          ON S > Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=Ø)
        END
    MIW: '-----------ØØØ---111ØØØØ-----------------------';
```

The above operation is used to set a test of the contents of register S. The NULL operands indicate that there is not transfer of data. It is assumed that subsequent control operations will test the ALU bits.

```
    EXP: COCYCLE
          LIT:= $L,
          NULL:= X'ØØØØØØ'&LIT AND DI
        END
    IMP: COCYCLE
     ON X'ØØØØØØ'&LIT AND DI = Ø DO (ALUZ:=1,NALUZ:=Ø,ALUNEG:=Ø),
     ON X'ØØØØØØ'&LIT AND DI < Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=1),
     ON X'ØØØØØØ'&LIT AND DI > Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=Ø)
        END
    MIW: '-----------Ø11Ø1111Ø1ØØØØ---ØØØ1Ø----LLLLLLL----';
```

The above operation is used to test a binary string against the contents of register DI. The use of the literal generator (LIT) provides for a wide range of strings.

```
  EXP: COCYCLE
         LIT:= $L,
         NCTR:= SELECT(Ø..7,LIT&X'ØØØØØØ')
        END
  IMP: COCYCLE
         ON LIT&X'ØØØØØØ' = Ø DO (ALUZ:=1,NALUZ:=Ø,ALUNEG:=Ø),
         ON LIT&X'ØØØØØØ' < Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=1),
         ON LIT&X'ØØØØØØ' > Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=Ø)
        END
  MIW: '----------ØØØ---111Ø11ØØ---ØØØ1Ø----LLLLLLLL----';
```

The above operation sets the counter to a binary value using bits Ø-7 of the word defined by "LIT&X'ØØØØØØ'". The SELECT defines the bits gated from the output of the ALU (Ø-7), while the expression defines a 32-bit word consisting

Of one byte generated by the literal generator (LIT), and 3 bytes defined as Ø's.

```
  EXP: NCTR:= NCTR - 1
  MIW: 'ØØØØ---Ø11--------------------------------------';

  EXP: T:= DI
  IMP: COCYCLE
         ON DI = Ø DO (ALUZ:=1,NALUZ:=Ø,ALUNEG:=Ø),
         ON DI < Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=1),
         ON DI > Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=Ø)
        END
  MIW: '-------------Ø11ØØØØ1111------------------------';

  EXP: T:= REGFILE[HIREG][$R] - 1
  IMP: COCYCLE
   ON REGFILE[HIREG][$R] - 1 = Ø DO (ALUZ:=1,NALUZ:=Ø,ALUNEG:=Ø),
   ON REGFILE[HIREG][$R] - 1 < Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=1),
   ON REGFILE[HIREG][$R] - 1 > Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=Ø)
        END
  MIW: 'ØØØØ------1ØØ---Ø1ØØ1111ØØØ---------RRRR---------';

  EXP: COCYCLE
         LIT:= $L,
         T:= X'FFFFFF'&LIT + T
        END
  IMP: COCYCLE
    ON X'FFFFFF'&LIT + T = Ø DO (ALUZ:=1,NALUZ:=Ø,ALUNEG:=Ø),
    ON X'FFFFFF'&LIT + T < Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=1),
     ON X'FFFFFF'&LIT + T > Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=Ø)
        END
  MIW: '----------1110000011111-------------LLLLLLL----';
```

The above four operations describe transfers involving the T register. The intent behind the last one is to substract a right-justified 8-bit quantity from the T register by using 2's complement representation. This, for example, the only way to decrement the T register, since the microengine does not provide a direct facility to do this. For example if we let LIT:= X'FF' we would substract 1 since the 2's complement of 1 is X'FFFFFFFF' in a 32-bit word. This is the kind of operation which shows that familiarity with the microarchitecture is an absolute requirement if one is to do meaningful microprogramming.

```
  EXP: S:= Ø
  MIW: 'ØØØØØØØ---------------------------------------Ø111';

  EXP: S:= S + T
  IMP: COCYCLE
         ON S + T = Ø DO (ALUZ:=1,NALUZ:=Ø,ALUNEG:=Ø),
         ON S + T < Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=1),
         ON S + T > Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=Ø)
        END
  MIW: '----------ØØØØØØØ11ØØØ1-------------------------';

  EXP: S:= S - T
  IMP: COCYCLE
         ON S - T = Ø DO (ALUZ:=1,NALUZ:=Ø,ALUNEG:=Ø),
         ON S - T < Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=1),
         ON S - T > Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=Ø)
        END
  MIW: '----------ØØØØØØ1Ø1ØØØ1-------------------------';
```

The above operations describe the setting of register S through various expressions. Notice that in S:= Ø there is no ALU involvement since this is achieved by a hardware clear operation.

```
  EXP: SHIFTC(RIGHT,1,DI(31),S&DI)
  MIW: 'ØØØØ-----------------------ØØØØ1--------1ØØ1----';
```

The above operation causes a doubleword right-circular shift on registers S and DI concatenated.

```
  EXP: REGFILE[HIREG][$R]:= DI
  IMP: COCYCLE
        T:= DI,
        ON DI = Ø DO (ALUZ:=1,NALUZ:=Ø,ALUNEG:=Ø),
        ON DI < Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=1),
        ON DI > Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=Ø)
       END;
       REGFILE[HIREG][$R]:= T
  MIW: '-------------Ø11ØØØØ1ØØØ------------RRRR--------';

  EXP: REGFILE[HIREG][$R]:= REGFILE[HIREG][$R] + 1
  IMP: COCYCLE
   T:= REGFILE[HIREG][$R] + 1,
   ON REGFILE[HIREG][$R] + 1 = Ø DO (ALUZ:=1,NALUZ:=Ø,ALUNEG:=Ø),
   ON REGFILE[HIREG][$R] + 1 < Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=1),
   ON REGFILE[HIREG][$R] + 1 > Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=Ø)
       END;
       REGFILE[HIREG][$R]:= T
  MIW: '----------1ØØ---Ø11Ø1ØØØ------------RRRR--------';

  EXP: REGFILE[HIREG][$R]:= S
  IMP: COCYCLE
        T:= S,
        ON S = Ø DO (ALUZ:=1,NALUZ:=Ø,ALUNEG:=Ø),
        ON S < Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=1),
        ON S > Ø DO (ALUZ:=Ø,NALUZ:=1,ALUNEG:=Ø)
       END;
       REGFILE[HIREG][$R]:= T
  MIW: '----------ØØØ---111Ø1ØØØ------------RRRR--------';
```

The operations above write to a register in the file, notice that they require two major clock cycles to complete. In the first one data is transferred to register T and the ALU bits are set, in the second cycle data is finally transferred from register T to the register in the file.

The timing constraints of testable conditions in the SEL are described as follows, using the syntax introduced in chapter three:

```
TEST ALUSIGN AFTER +1;
     ALUZ AT +2;
     NALUZ AT +2
     ALUNEG AT +2
     NCTR = Ø AFTER +1
     BIBUSY AFTER +1
```

The above description determines that ALUZ, ALUNEG, NALUZ can be tested for the value set in the previous second microinstruction and only in that particular microinstruction, while the others can be tested for a value set at least in the previous second microinstruction. These timing constraints originate in the fact that the SEL uses pipelining in the processing of microinstructions, i.e., the clock cycles of contiguous microinstructions overlap.

Some control operations in the SEL are described below. Many of them are conceptually the same except for differences in the range of addresses which they can reach:

```
CONTROL

 EXP: GOTO $A
 IMP: COCYCLE
       MPC:= MPC + 1,
       MPC(9..12):= $A
      END
 MIW: 'ØØØØ1ØØ--------------------------------------AAAA';

 EXP: GOTO $A
 IMP: COCYCLE
       MPC:= MPC + 1,
       MPC(5..12):= $A
      END
 MIW: 'ØØØØ1Ø1----------------------------------AAAAAAAA';

 EXP: GOTO $A
 IMP: COCYCLE
       MPC:= MPC + 1,
       MPC(1..12):= $A
      END
MIW: 'ØØØØ11Ø-----------------------------AAAAAAAAAAAA';

EXP: GOTO $A
IMP: COCYCLE
       MPC:= MPC + 1,
       MPC:= $A
      END
MIW: 'ØØØØ111----------------------------AAAAAAAAAAAAA';
```

Notice that the above instructions are conceptually the same, i.e., an unconditional branch, but they have different addressing ranges. Addresses are formed by the microcode generator by incrementing the microprogram counter by 1 and sourcing the low order bits from the microword. The four operations above address respectively within a 16-location absolute range, a 256-location absolute range, a 4Ø96-location absolute range, and a 8192-location absolute range.

There are certain things about the SEL which should not be included in the HAL description. These relate to the conventions used in microprogramming and to arbitrary values permanently contained in certain storage items. An example of the first is the status of the microengine upon entry into a user microprogram. Among other things the conventions indicate that the HIREG bit is set to Ø indicating access to the lower bank of the register file. All items can be manipulated at will, but some, such as the program counter (PC) must be restored upon exit from the user microprogram.

Another example of conventions is the location of program status words. These come in two modes: program status word (PSW) mode and program status doubleword (PSDW) mode. For these two consecutive words in the scratch file (SCRATCH) have been reserved. The particular mode being used can be determined by inspecting the first word. The formats of the two modes can be described using HAL notation as follows:

```
TYPE PSW= TUPLE
            PRIV: BIT;
            CCODES: SEQ (1..4) OF BIT;
            EA,BIT6: BIT;
            BIT7TO12: SEQ (7..12) OF ZERO;
            PC: SEQ (13..29) OF BIT;
            C: BIT;
            BIT31: ZERO
           END;

     PSDW= TUPLE
            WORD1: TUPLE
                    PRIV: BIT;
                    CCODES: SEQ (1..4) OF BIT;
                    EXT,RHW,AEXP,PSD,MAP: BIT;
                    PCOUNTER: SEQ (10..29) OF BIT;
                    NR, BLK: BIT
                   END;
            WORD2: TUPLE
                    GRAN: SEQ (32..33) OF BIT;
                    BPIX: SEQ (34..45) OF BIT;
                    BIT46: ZERO;
                    RET: BIT;
                    EXTINTFLAG: SEQ (48..49) OF BIT;
                    CPIX: SEQ (50..61) OF BIT;
                    BIT62TO63: SEQ (62..63) OF ZERO
                   END
           END;
```

Since there is not a separate hardware item to contain status information, one can not include the above items in the HAL description of the SEL.

An example of arbitrary values contained in a storage item is register 15 of the upper bank in the register file. This register is permanently set to zero and even though the microengine will process a write operation to it, no actual data transfer occurs.

The intent of this chapter has been to the demonstrate the ability of HAL to describe existing storage structures and operations in a real machine. A complete description of the SEL32/75 would require considerably more space, given that the SEL is a rather complex machine. The partial description used to code sample microprograms in this thesis is shown in appendix C.

4.2) THE AM2900 FAMILY

The AM2900 family consists of a series of LSI building blocks designed for use in microprogrammed computers and controllers. Each device is designed to be expandable and sufficiently flexible to be suitable for emulation of many existing machines.

Figure 4.3 illustrates a typical system architecture. There are two sides to the system. At the left is the control circuitry and on the right is the data manipulation circuitry. At the center of the system is the pipeline register which contains the microinstruction currently being executed. Each microinstruction contains not only bits to control the data hardware, but also bits to define the location in PROM of the next microinstruction to be executed. The I field indicates to the sequencer the source of the next address. The CC field indicates to the sequencer the conditions under which the I field applies. The BA field provides a branch address if necessary.

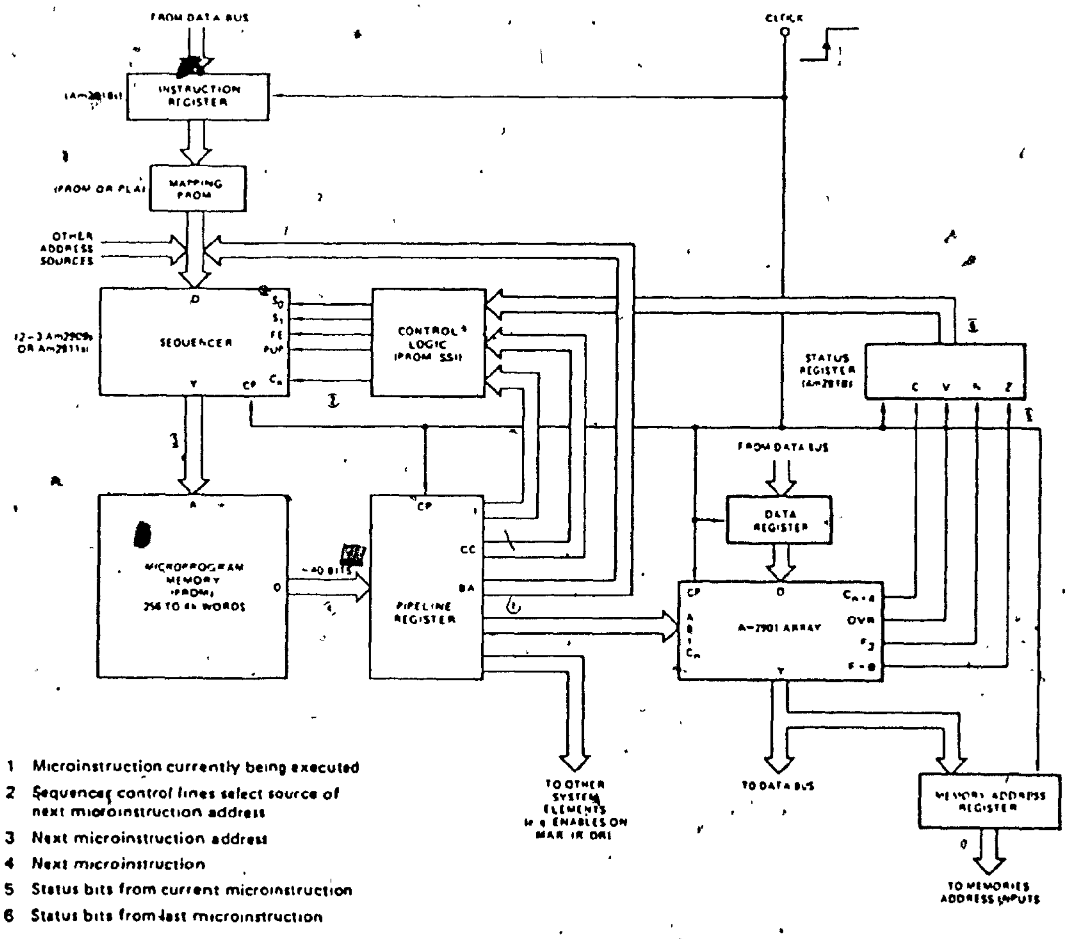

FROM DATA BUS
INSTRUCTION REGISTER
(Am2918s)
MAPPING PROM
(PROM OR PLA)
OTHER ADDRESS SOURCES
SEQUENCER
(2-3 Am2909s OR Am2911s)
D
S0
S1
FE
PUP
Cn
Y
CP
CONTROL LOGIC (PROM SSI)
A
MICROPROGRAM MEMORY (PROM) 256 TO 4k WORDS
O
~40 BITS
PIPELINE REGISTER
CP
I
CC
BA
CLOCK
STATUS REGISTER (Am2918)
C
V
N
Z
FROM DATA BUS
DATA REGISTER
CP
D
A
B
I
Cn
Am2901 ARRAY
Cn+4
OVR
F3
F=0
Y
TO OTHER SYSTEM ELEMENTS (e.g. ENABLES ON MAR OR DR)
TO DATA BUS
MEMORY ADDRESS REGISTER
O
TO MEMORIES ADDRESS INPUTS
1 Microinstruction currently being executed
2 Sequencer control lines select source of next microinstruction address
3 Next microinstruction address
4 Next microinstruction
5 Status bits from current microinstruction
6 Status bits from last microinstruction

Figure 4.3 Typical AM29ØØ System (from BELL82)

The centerpiece of the data manipulation hardware is AM29Ø1 array. This is constructed using one or more AM29Ø1 4-bit slice microprocessors. The device as shown in figure

4.4 consist of a 16-word by 4-bit RAM, a high speed ALU, a shift register, and the associated decoding and multiplexing circuitry. The microprocessor is cascadable with full lookahead or with ripple carry, has three-state outputs, and provides various status flags from the ALU.

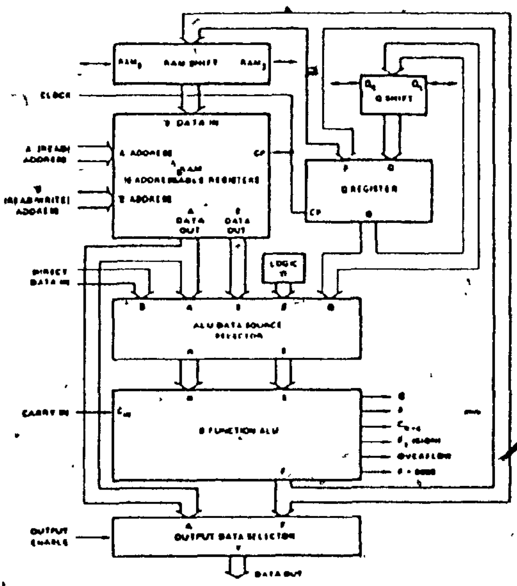


Figure 4.4 AM2901 Architecture (from BELL82)

To describe an AM2901 array the approach should be the description of the end result of the array, and not of the individual components. For example the array of three AM2901 microprocessors shown in figure 4.5 yields a data path 12-bit wide. The RAM file can then be treated as 16-word by 12-bit, and the shift register Q can now be considered as a 12-bit register.

The basic type would be described as follows:

```
TYPE WORD = SEQ (11..Ø) OF BIT;
```

The storage items would be described as shown below:

```
STORE RAMFILE: ARRAY [Ø..15] OF WORD;
      Q: WORD;
```

The status bits of the ALU can be described as follows:

```
       ALUZ,ALUSIGN,OVRF,COUT: BIT;
```

ALUZ is set to 1 if the result of the ALU operation is zero. ALUSIGN contains the most significant bit of the ALU's result. OVRF is set to 1 if an overflow results from the ALU's operation. COUT represents the carry-out from the ALU's operation.

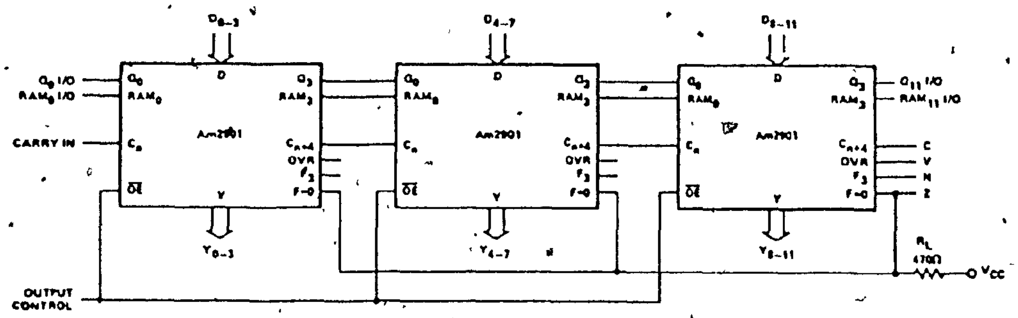


Figure 4.5 Cascaded AM29Ø1's (from BELL82)

Some typical operations in the AM29Ø1 are shown below:

```
EXP: RAMFILE[$B]:= RAMFILE[$A] + RAMFILE[$B]
IMP: COCYCLE
       ON RAMFILE[$A] + RAMFILE[$B] = Ø DO (ALUZ:= 1),
       ALUSIGN:= SELECT(11,RAMFILE[$A] + RAMFILE[$B]),
       ON RAMFILE[$A] + RAMFILE[$B] > 4095 DO (OVRF:= 1)
     END
```

```
MW: '----0010-011-001-000AAAABBBB----';

EXP: Q:= RAMFILE[$A]
IMP: COCYCLE
      ON RAMFILE[$A] = 0 DO (ALUZ:=1),
      ALUSIGN:= RAMFILE[$A](11)
     END
MW: '----0010-000-100-011AAAA--------';
```

The first of the above two operations describes the addition of 1 word in the RAM file to another. The second operation is a straight transfer of a word in the RAM file to the Q register.

The am2901 is capable of performing shift operations througt the control of the ALU's destination. Shown below is a double arithmetic shift:

```
EXP: COCYCLE
      RAMFILE[$B]:= SHIFTV(RIGHT,1,ALUSIGN,RAMFILE[$B]),
      SHIFTC(RIGHT,1,RAMFILE[$B](0),Q)
     END
IMP: COCYCLE
      ON RAMFILE[$B] = 0 DO (ALUZ:=1),
      ALUSIGN:= RAMFILE[$B](11)
     END
MW: '----001011111011-011----BBBB----';
```

The timing of tests in the AM2901 would be described as shown below:

```
TEST ALUSIGN AT +1;
     ALUZ AT +1;
     OVRF AT +1;
     COUT AT +1;
```

The sequencing hardware of figure 4.3 is usually constructed with AM2909's or AM2911's sequencers. Figure 4.6 shows the architecture of an AM2909 sequencer. This component can select an address from four possible sources: a set of external direct inputs (D); external data from the R inputs, stored in an internal register; a 4-word deep push/pop stack; a program counter register, usually containing the last address plus one. Several AM2909's can be interconnected to produce a larger address width and allow a larger address range.

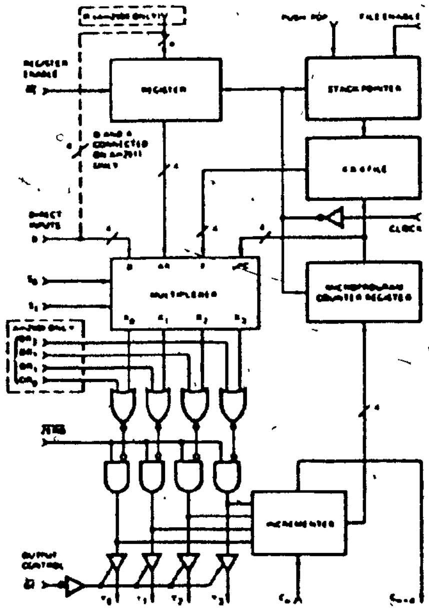


Figure 4.6 AM2909 Architecture (from BELL82)

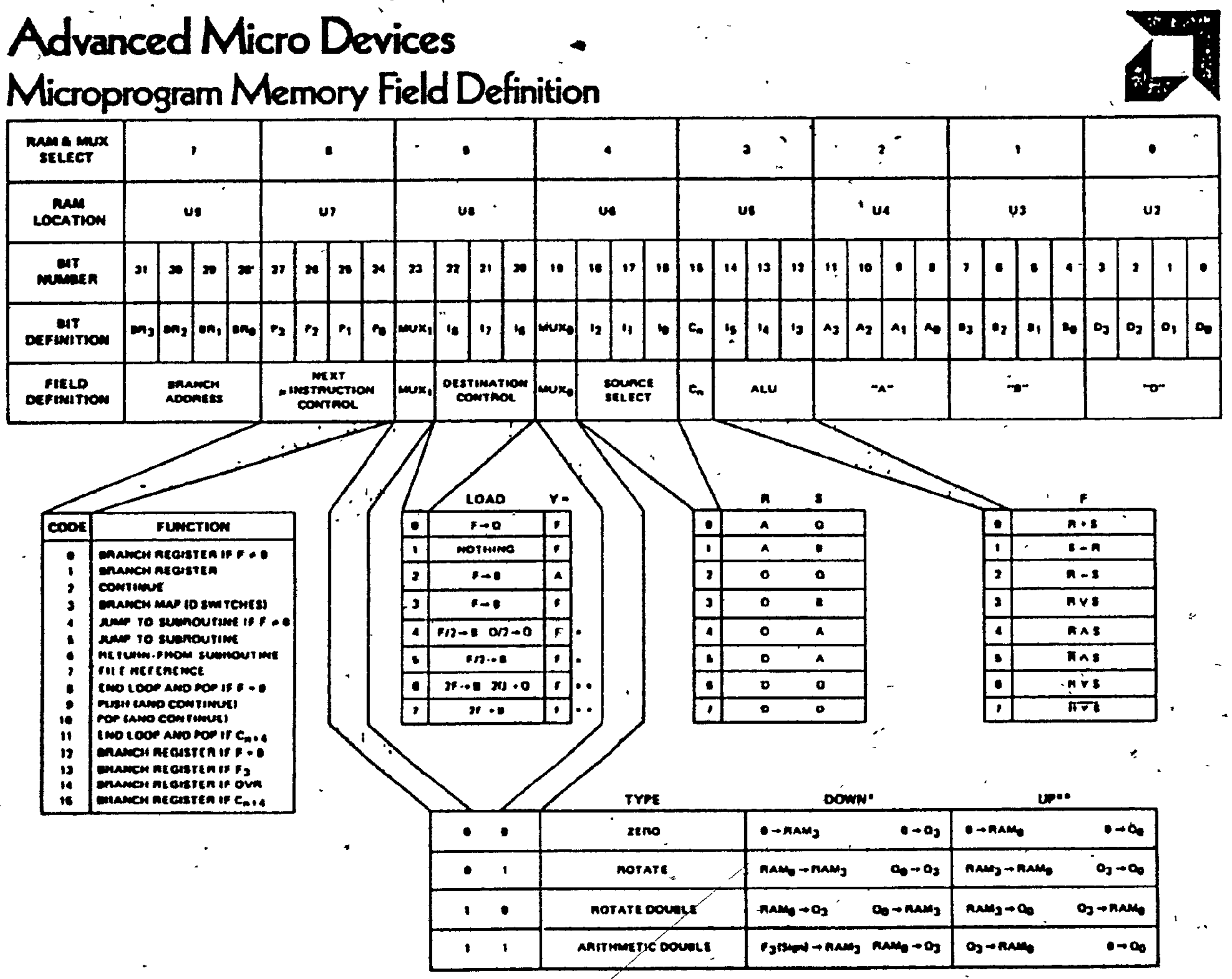
Some of the control operations of which the AM2900 is capable are shown below:

```
EXP: ON ALUZ GOTO BREG
IMP: COCYCLE
       ON ALUZ DO (PIPEREG:= PROM[BREG], MPC:= BREG + 1),
       ON NOT ALUZ DO (PIPEREG:= PROM[MPC], MPC:= MPC + 1)
     END
MW: '----0000-------------------------------';

EXP: GOSUB $A
IMP: COCYCLE
       PIPEREG:= PROM[$A],
      PUSH(MPC,ASTACK),
      MPC:= $A + 1
     END
MW: 'AAAA0101---------------------------------';
```

the first of the two above operations describes a conditional branch to the address contained in the branch register. The second describes an unconditional jump to a subroutine. PROM refers to programmable ROM where microprograms are stored. PIPEREG referes to the pipeline register where microinstructions are decoded. ASTACK refers to hardware stack available in the AM2909. MPC refers to the microprogram counter.

The microword format for the AM2900 systems is shown in figure 4.7. This is the basic format for a 4-bit slice. The address field of this format is subject to expansion if a larger address range is desired.



Advanced Micro Devices

Microprogram Memory Field Definition

RAM & MUX SELECT	7				6				5				4				3				2				1				0			
RAM LOCATION	U9				U7				U8				U6				U5				U4				U3				U2			
BIT NUMBER	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIT DEFINITION	BR_3	BR_2	BR_1	BR_0	P_3	P_2	P_1	P_0	MUX_1	I_8	I_7	I_6	MUX_0	I_2	I_1	I_0	C_n	I_5	I_4	I_3	A_3	A_2	A_1	A_0	B_3	B_2	B_1	B_0	D_3	D_2	D_1	D_0
FIELD DEFINITION	BRANCH ADDRESS				NEXT μ INSTRUCTION CONTROL				MUX_1	DESTINATION CONTROL			MUX_0	SOURCE SELECT			C_n	ALU			"A"				"B"				"D"			

CODE	FUNCTION
0	BRANCH REGISTER IF F ≠ 0
1	BRANCH REGISTER
2	CONTINUE
3	BRANCH MAP (D SWITCHES)
4	JUMP TO SUBROUTINE IF F ≠ 0
5	JUMP TO SUBROUTINE
6	RETURN FROM SUBROUTINE
7	FILE REFERENCE
8	END LOOP AND POP IF F = 0
9	PUSH (AND CONTINUE)
10	POP (AND CONTINUE)
11	END LOOP AND POP IF C_{n+4}
12	BRANCH REGISTER IF F = 0
13	BRANCH REGISTER IF F_3
14	BRANCH REGISTER IF OVR
15	BRANCH REGISTER IF C_{n+4}

	LOAD	Y =	
0	F → Q	F	
1	NOTHING	F	
2	F → B	A	
3	F → B	F	
4	F/2 → B Q/2 → Q	F	*
5	F/2 → B	F	*
6	2F → B 2Q → Q	F	**
7	2F → B	F	**

	R	S
0	A	Q
1	A	B
2	0	Q
3	0	B
4	0	A
5	D	A
6	D	Q
7	D	0

	F
0	R + S
1	S − R
2	R − S
3	R ∨ S
4	R ∧ S
5	$\overline{R} \wedge S$
6	R ∀ S
7	$\overline{R \forall S}$

		TYPE	DOWN*		UP**	
0	0	ZERO	$0 \rightarrow RAM_3$	$0 \rightarrow Q_3$	$0 \rightarrow RAM_0$	$0 \rightarrow Q_0$
0	1	ROTATE	$RAM_0 \rightarrow RAM_3$	$Q_0 \rightarrow Q_3$	$RAM_3 \rightarrow RAM_0$	$Q_3 \rightarrow Q_0$
1	0	ROTATE DOUBLE	$RAM_0 \rightarrow Q_3$	$Q_0 \rightarrow RAM_3$	$RAM_3 \rightarrow Q_0$	$Q_3 \rightarrow RAM_0$
1	1	ARITHMETIC DOUBLE	F_3(Sign) $\rightarrow RAM_3$	$RAM_0 \rightarrow Q_3$	$Q_3 \rightarrow RAM_0$	$0 \rightarrow Q_0$

Figure 4.7 The AMD2900's Microword Format

CHAPTER 5: THE PROCESS OF MICROPROGRAMMING

In chapters three and four we were concerned with describing a microprogrammable computer in terms which an application programmer could easily understand. At no point was a reference made to a possible practical application. Since this is the goal of any programming system, tools must be provided to aid in the generation of good microcode. The notation used to specify a microprogram is referred to as a microprogram specification language and it is the main subject of this chapter.

In order to illustrate the process of microprogramming, I would like to draw an analogy between a calculator and a microprogrammable computer. Consider a programmer who has a certain algorithm to implement. If he were to use a calculator, he would first become acquainted with the calculator, its storage structure, the operations it can perform and the buttons that carry these out. He would then proceed to select some storage items for his data and to establish the necessary sequence of operations to obtain the desired results. Operations are invoked by pressing buttons. The order and the circumstances under which operations are invoked are determined by the programmer himself as the algorithm proceeds.

There are some similarities between the operation of a calculator and that of a microprogrammed computer.

Microinstructions are directly executable since they carry out physical actions without using any intermediate code. The microengine can therefore be treated as a hardware interpreter. Microinstructions are similar to the buttons of a calculator in the sense that when presented to the microengine certain actions are performed, just as when a button is pressed in the calculator. Horizontal microinstructions are capable of multiple operations but this is just a difference in implementation. The buttons in a calculator would correspond more exactly to the extreme vertical format of a microinstruction in which only one operation is performed. The storage structure of a microprogrammed computer is considerably more complex than that of a calculator but this again is a difference in implementation.

There are, of course, some major conceptual differences between a calculator and a microprogrammed computer, the most important being the fact that a calculator has no control circuits that the programmer can use. In the calculator control is exercised by the programmer himself when he decides which button to press next. Another major difference is that one can not store a program in a calculator while a microprogrammed computer has a control storage available to store microinstructions and the necessary circuitry to sequence them [1].

1 Programmable calculators are considered hand-held computers and not calculators in the strict sense

Microprogramming can then be considered as the process of operating a microprogrammed computer. A microprogram is a discrete set of operations acting on a discrete set of storage resources which are executed under specific circumstances and in a particular order.

A programmer considering the implementation of an algorithm as a microprogram would proceed in a similar manner as if he were to use a calculator. He would get acquainted with the microarchitecture, its storage structure, and the operations which can be performed. He would then select some storage items for his data and determine the necessary operations to achieve the desired results. Besides functional operations, the programmer also considers control operations to establish the circumstances and the order in which the operations are to be carried out.

5.1) THE MICROPROGRAMMING ENVIRONMENT

In order to produce useful microcode, an institution having a user-microprogrammable computer must establish an environment that is conducive to good microprogramming practices. The prospective microprogrammer must be made aware that he will deal with the computer at a lower, more detailed level. The potential for great accomplishments or disastrous failures is very great, since at the

because they offer program memory and control instructions, although in a somewhat primitive form.

microprogramming level all the resources of the computer are visible and available for use.

In order to provide this environment I have proposed a Comprehensive MicroProgramming System (CMPS) which deals with the various aspects of the microprogramming process, namely:

1) Familiarization with the microarchitecture.

2) microprogram specification.

3) microprogram correctness.

4) microcode generation.

5) Control storage management.

The experimental work of this thesis has been concentrated in phases 1,2, and 4 of the above list. The elements of the system are described below.

Familiarization with the microarchitecture means getting acquainted with the computer at the microprogramming level, its resources and capabilities.

Microprogram specification is a detailed description of the objectives of the microprogram and of the requirements to achieve them. The requirements usually include a description of the storage resources needed and of the operations to be performed. This phase of the process requires the existence of a notation or language to specify the microprogram [DAVI80].

Microprogram correctness determination involves verification, testing and debugging. Davidson & Shriver describe these as follows [DAVI80]: "by verification we mean the attempt to prove the absence of errors from the program or microprogram. By testing we mean the attempt to discover errors in the code or microcode, and the measurement of how well the specifications of the problem have been met. By debugging we mean the location and correction of known errors in the code or microcode".

Microcode generation involves the processing of a microprogram specification in order to produce a sequence of executable microwords. This process involves the composition of microoperations into single microinstructions to fully utilize the concurrency of the machine and optimize the microcode.

Control storage management refers to the administration of the memory space used to hold microprograms. There are various aspects to this task, such as space allocation, microcode loading and location, and long term secondary storage of microcode. The management of control storage may become more complex as the technology improves to allow dynamic microcode relocation and virtual control storage, Guha has already described a dynamic microprogramming system [GUHA77]. In any installation having a user-microprogrammable computer it is appropriate to designate a Writeable Control Storage (WCS) administrator to allocate space, to maintain and provide upto date information, and to avoid duplication of microcode.

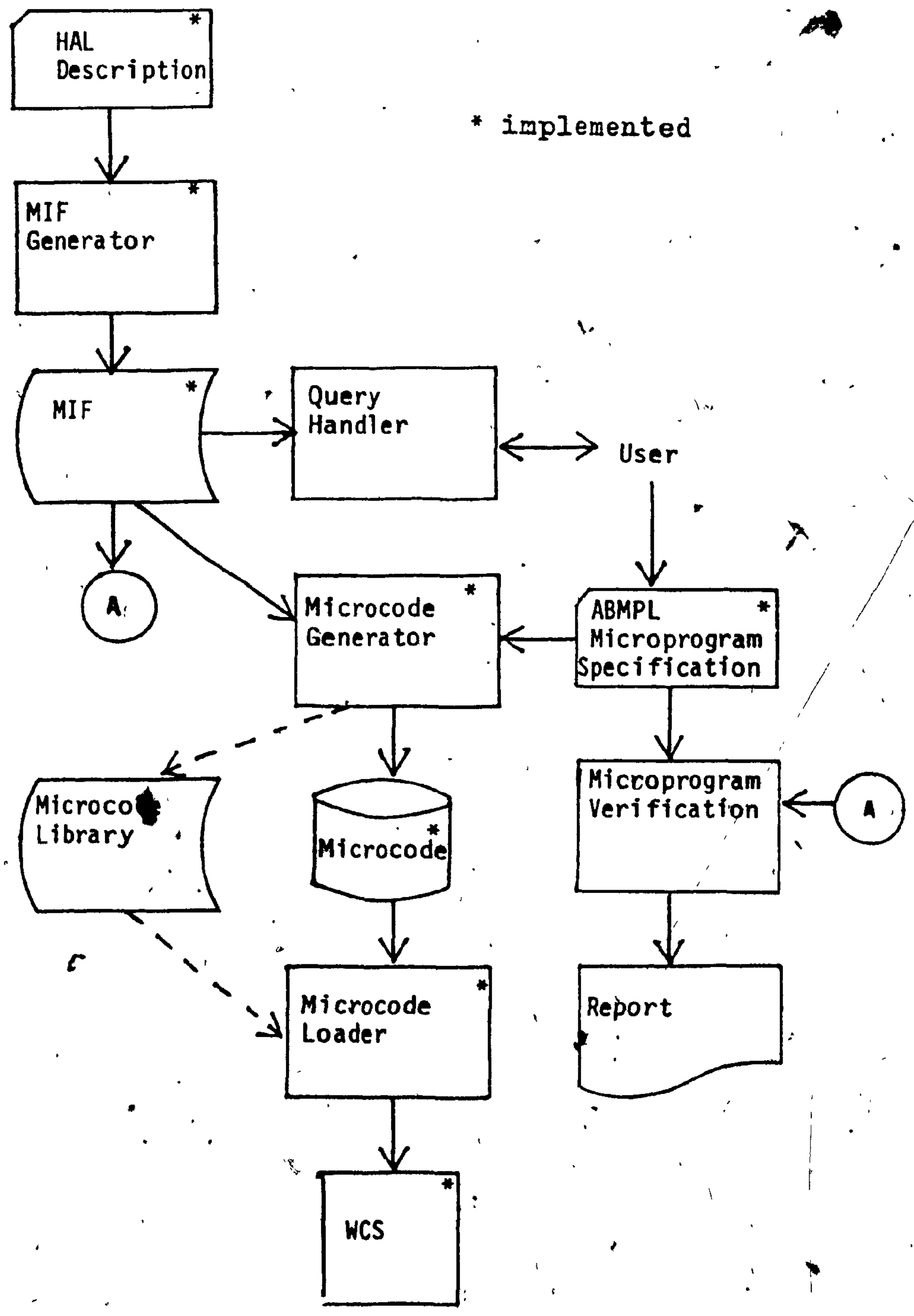


Figure 5.1 Comprehensive MicroProgramming System

An overview of CMPS is shown in figure 5.1. The fundamental components are

An overview of CMPS is shown in figure 5.1. The fundamental components are the Machine Information File (MIF) and the microprogram generator. This design addresses the various phases of the microprogramming process. The MIF is generated from a HAL description of the microarchitecture. A user learns about the machine by querying the MIF. The user codes a microprogram using a high-level notation (here ABMPL). This specification is then processed by the microcode generator to produce code which can then be loaded into WCS or stored in a library for later use.

5.1.1) GETTING ACQUAINTED WITH THE COMPUTER

The Machine Information File is the centerpiece of CMPS. It is used by the various phases of the microprogramming process and in particular by the microprogrammer to familiarize himself with the microarchitecture.

The MIF is generated by processing the HAL description of the machine. This is accomplished by subjecting the HAL description to a compilation process. The compiler verifies all the storage references and parses all the constructs to certify that they are syntactically correct. This compiler will not yield executable code, but it generates records for the MIF.

The MIF should be organized as a database to provide fast and convenient access to machine information. The structure of the MIF should resemble that of the HAL description, that is, it could be organized in four major segments: data structure, functional operations, tests, and control operations.

The data structure segment would be nothing more than the symbol table used by the compiler. The functional operations could be parsed into records with a code to indicate the type of operation and another code to identify the particular operation. Each record would contain explicit as well as implicit information, the microword associated with the operation, and any necessary information to resolve the use of literals at microcode generation time.

The overriding consideration in the organization of the MIF must be the fast and convenient access to the information. The prospective microprogrammer will access the MIF through a query program which will treated it as a read-only file. The query program must be very flexible to accommodate general or specific user-defined search criteria.

A typical operation on the MIF would be a query to display the data structure of the machine. Attention can then be focused on a particular item, and further queries may be used to display the operations which can be performed on this item.

Davidson & Shriver [DAVI80] describe a similar microprogram support system used for educational purposes at the university of Southwestern Louisiana. This system includes a microprogram simulator which has HELP facilities to display documentation about the resources of the machine. In CMPS the facilities to document machine resources are separated from any other component in the system. Documentation in CMPS will be done through the use of a query program on the Machine Information File.

For the purpose of this thesis a small compiler has been constructed. It processes a HAL description and yields a sequential file of ordered records which is used by the microcode generator. Each record is identified by a code having two components, one is the record type indicating whether it represents a storage item, a functional operation, a test or a control operation. The other component is a numeric code which uniquely identifies the record, this code is generated from character strings used in the HAL description of the item.

The format of the records varies according to type. In the case of functional or control operations the records contain the microword and any necessary information to resolve the use of literals at microcode generation time. Test records contain the necessary information to establish the order of testing at microcode generation time.

The full implementation of the MIF as a database is a project complex enough to deserve a separate undertaking. Since the HAL description of a machine establishes implicit relationships between storage items and the operations which can be performed on them, a relational implementation would seem most appropriate.

Richard & Lewis [RICH8Ø] describe a microinstruction as a unique set of microoperations, and each microoperation as a quintuple <OP,I,O,F,P> where OP= function (+,-,*, etc.), I= input data set, O= output data set, F= microword field and P= clock phase. In our case the F component is not required since the complete microword is given and the mapping of resources to the microword is implicit. This approach could be used in organizing the MIF. We would then have a set of microinstructions which consist of a unique set of microoperations, which in turn are composed of various sets of resources. The necessary information can be gathered from the HAL description of the machine.

The intention here has been to show the feasibility and usefulness of such a file in a microprogramming support system. The field of computer science has developed very powerful tools for the management of information, and nothing is more appropriate than to use such tools to manage information about computers.

5.1.2) THE SPECIFICATION OF A MICROPROGRAM

Once the programmer has determined the storage and operational needs of his algorithm he needs a notation to describe under what circumstances and in what order the operations he selected are going to be performed on the storage items he is using. For this purpose I have designed a microprogram specification language called Alternative Based Microprogramming Language (ABMPL) which specifies a microprogram by describing an abstract machine (a subset of the microengine) and an abstract program.

ABMPL is modelled after the ABL (Alternative Based Language) programming methodology [JAW081] which treats a program as a set of requirements of the form: "when a certain situation arises, perform an associated function". Such a requirement is called an alternative, the certain situation a predicate(s) and the associated function as a sequence of actions.

Berg & Franta have described the life cycle of a firmware engineering system as having the following stages [BERG80]:

REQUIREMENTS ENGINEERING: identifies the major functional requirements and attributes of the system.

NON-PROCEDURAL DESIGN: formalizes the functional requirements and the attributes given by requirements engineering; the result is a functional specification and a property specification.

PROCEDURAL DESIGN: uses the specification to produce blueprints for programs to be implemmented.

IMPLEMENTATION: generates programs that embody the blueprints provided by procedural design.

INTEGRATION: coordinates programs so that assemblage results in an entire program system.

OPERATION: executes the program system such that its effects can be observed.

VERIFICATION: demonstrates the verisimilitude of procedural design, implementation, integration, and operation with system specification.

DOCUMENTATION: describes the results of all stages of the life cycle.

MAINTENANCE: preserves the operational status of the system including extensions, corrections, modifications, etc..

The above stages are represented in the various components of CMPS. Procedural design, implementation and integration are found in the microcode generator which uses the HAL description of the machine as contained in the MIF. Documentation and maintenance involve both the microprogrammer and the WCS administrator. ABMPL is concerned with requirements engineering and, to a certain extent, with non-procedural design, and it is in these two areas where Berg & Franta state that very little progress has been made [BERG80].

The choice of an ABL-type notation such as ABMPL, as opposed to a strictly procedural specification such as existing high-level microprogramming languages like EMPL [DEWI76], was made based on the following considerations:

1) ABL is a programming technique that supports separation of the evaluation of data (control operations) from the modification of data (functional operations) [JAWO81]. This is an important characteristic, since Dasgupta has shown that the sequencing mechanism is a great source of variability [DASG80], and also Jones has shown that existing microprogrammable computers do not support the implementation of high-level control constructs [JONE75] which would be required in a strictly procedural specification. Using ABMPL, a programmer is able to concentrate on the functional aspects of his microprogram and to express control in very simple abstract terms. The microcode generator processes the abstract control specification and implements actual control using the control operations described in the MIF. This provides a degree of machine independence since the programmer does not need to be aware of the sequencing mechanism of a particular machine.

2) The ABL programming technique establishes a sequence of actions associated with each alternative. These sequences are branch-free segments of code which constitute the basis of any optimization process [LAND80]. Furthermore, this technique allows the use of sequencing operators (','

parallel execution, ";" serial execution) to permit or inhibit the optimization process. In this way the microprogrammer aids the optimization process, an idea supported by Dasgupta through the use of REGION blocks to inhibit optimization [DASG80], and by Malik & Lewis through the use of programmer provided look-ahead information for the generation of microcode [MALI78].

3) Since ABL imposes a well defined structure on a program, the information provided can be very easily stored and manipulated to produce very sophisticated display formats [JAWO81]. These not only provide an excellent documentation, but can be used to analyze resource usage (data flow) and instruction sequences (for manual optimization). The task of maintenace is simplified since it is very easy to determine the effect of a modification on a program.

4) An ABL specification lends itself very well to a process of verification by simulation since it establishes the storage needs and behavior of a program. An ABL specification determines an abstract machine by defining storage items (variables), a set of possible actions (instruction set), and a set of relevant predicates (testable conditions). An abstract program is also established to determine the order and the circumstances under which the actions are to be executed. The problem is then to simulate the abstract machine on a real machine, and to run the abstract program to see its results. A process

of interpretation seems the most suitable solution and indeed this has been the route taken by Jaworski and Hinterberger [JAW081].

5) The structure of an ABMPL microprogram specification is in line with the overall strategy of this thesis, i.e., to use abstractions of existing entities in order to produce microprograms. In defining an abstract machine, the microprogrammer is able to deal with a simpler machine which is a subset of the real machine. In specifying an abstract program, the microprogrammer establishes the behavior of the microprogram without any reference to the machine dependent sequencing mechanism.

ABMPL defines a microprogram as an abstract machine and an abstract program. The general frame of an ABMPL program is shown below:

```
MPROGRAM  ......(.........);         (*heading*)

 STORE  ......, ......, .......;     (*store *)

 SYN    ......=....;             (*synonyms*)
        .....=....;

 MPROCEDURE .........;            (*microprocedure*)

 ISET                               (*operations*)
  1) ...........;
  2) .........;

 PREDICATES                       (*testable conditions*)
  1) .........;
  2) ........;

 PROGRAM                       (*abst program*)
  CLUSTER 1
   1) ......... .

  CLUSTER 2
```

```
   1) ........
   2) ......
END.
```

The complete grammar of ABMPL is given in appendix A, but I will discuss below the most important aspects. In BNF we have:

```
<microprogram> ::= MPROGRAM <identifier>(<base address><parm>);
                    <machine>
                    <program>
                   END.
```

the base address must be provided to indicate where the microcode is to be loaded. The WCS administrator will inform the microprogrammer of the memory space available in WCS. <parm> is used to document the use of a microprogram as a subprogram. This parameters list is used only for information purposes. At the microprogramming level there is no formal parameter passing because the overhead is prohibitive [DEWI76]. The abstract machine describes the resources used in the microprogram, storage items, operations and the testable conditions. In describing the abstract machine the microprogrammer determines a smaller machine which is imbedded in the real machine. In BNF we have:

```
<machine> ::= <storage>
              <synonyms>
              <microprocedures>
              <instruction set>
              <predicates>
```

<storage> represents a list of the storage items being used. The items listed must exist in the MIF. Synonyms are a

facility used to rename existing storage items for user convenience. Properly used synonyms may provide a certain degree of portability for microprograms. For example consider the use of the S register of the SEL32/75 in shifting operations, the S register and its synonym could be declared as follows:

```
    STORE S;
    SYN SHIFREG = S;
```

then we would use SHIFREG instead of S throughout the microprogram. If one transports this microprogram to another computer then one would look for a register with characteristics similar to those of the S register, declare the new register as storage and define SHIFREG as its synonym. In this way the rest of the microprogram is not affected.

Microprocedures in ABMPL fill the same function as in any other language, they can be of two types: internal and external. Internal microprocedures are defined within the microprogram and will form part of the generated microcode, their structure is similar to that of a microprogram. External microprocedures are assumed to exist separately and their address is provided for branching purposes. This facility helps avoid duplication of microcode since the WCS administrator would mantain a list of commonly used microprocedures. In BNF we have:

```
<microprocedure> ::= MPROCEDURE <identifier><parameters>;
                     <microprocedure block>
```

```
                    END;
<microprocedure block> ::= <machine> <procedure> !
                           <EXT <base address>
```

The instruction set describes the operations of which the abstract machine is capable. Any operation listed here must be syntactically correct and exist in the MIF. In BNF we have :

```
<instruction set> ::= ISET <instruction list> !
                      <empty>

<instruction> ::=
 <i code>) <valid operation> <implicit assignment>
```

the instruction code is a mnemonic used for referral purposes in the abstract programs or procedures. <valid operation> has a syntax similar to the explicit operations in HAL and it must describe an operation existing in the MIF. <implicit assignment> is used to document the fact that some storage items are implicitly affected by this operation and that such items are relevant for testing purposes. Malik & Lewis support this idea when they discuss the implicit setting of flags during ALU operations [MALI78]. The way in which the storage items are affected is described in the implicit operations associated with the explicit operation being used.

The predicates describe the list of testable conditions which are relevant for this microprogram. In BNF we have:

```
<predicates> ::= PREDICATES <predicate list>
<predicate> ::= <predicate code>) <expression>
```

the predicate code is a mnemonic used for referral purposes in the abstract programs or procedures. The expression describes a testable condition and must exist int the MIF.

Abstract programs and procedures are defined as a list of clusters. The position of a cluster in the list is irrelevant except for the first cluster which is considered the entry point of the program. Each cluster in turn is a set of alternatives. The alternative consists of a set of predicates guarding an associated list of actions. In addition, each alternative includes information to indicate which cluster, if any, should be considered next. In BNF we have:

```
<program> ::= PROGRAM <cluster list>

<cluster> ::= CLUSTER <cluster code> <alternative list>

<alternative> ::= <alternative code>) <abstract statement>

<abstract statement> ::=
 <abstract conds> <abstract actions> <abstract control>

<abstract conditions> ::= ON <predicate references> !
                         <empty>

<abstract actions> ::= DO <instruction references> !
                      <empty>

<abstract control> ::= GO <cluster code> !
                      RETURN
```

Predicate references are a list of conditions which must be met in order to execute the associated actions. The '+' or '-' signs in front of a predicate code are used to indicate if the condition must be true or false. Instruction references are a list of instruction codes separated by a sequencing operator to permit parallel execution (',') or to force serial execution (';'). The RETURN keyword indicates a halt in the execution of the program or procedure and a return to the level above from which it was called. This reflects the structure in which microprograms are called by a supervisor microprogram which handles the general fetch-decode-execute cycle of machine instructions.

A microprogram described using ABMPL can be considered as an abstract microprogram. Giloi et al [GILO80] offer a definition of an abstract microprogram which is consistent with the goals of ABMPL: "an abstract microprogram is an operational specification of a machine operation (instruction) performed on certain objects of the machine". Furthermore, they state that the language used to write abstract microprograms must feature data abstraction, operation abstraction, and control abstraction.

ABMPL offers all three of the above features. In particular, Giloi et al's definition of operation abstraction is very consistent with that of ABMPL. By operation abstraction they mean that "the language must contain a comprehensive set of elementary operations which are primitive and axiomatic in the sense that, (1) there will exist a functional unit in the hardware for its execution and, (2) the operation's behavior is pre-defined, together with the representation of the objects the operation may be applied to".

For control abstraction Giloi et al suggest the use of high-level constructs such as WHILE-DO, REPEAT-UNTIL, etc., since they are concerned with a strictly procedural specification of microprograms. It must remembered, however, that such control constructs have been shown difficult to implement in the microarchitecture. ABMPL also offers control abstraction but in a simpler form through the specification of next cluster information.

5.1.3) THE OPERATION OF CMPS

CMPS is to be operated as a software aid to microprogramming. The WCS administrator would be in charge of the maintenance of the system. Ideally the HAL description of the machine should be done by the manufacturer, but given time and resources this task could be performed locally.

Initially the programmer will outline his microprogram, and then contact the WCS administrator for information on the access to CMPS. The programmer will then proceed to work with CMPS. The steps are listed below:

1) Analysis of the Microarchitecture. In this step the programmer accesses the MIF to identify the resources required by his program. The programmer selects storage items, functional operations, and testable conditions according to the requirements of his algorithms. In effect

the programmer constructs a machine suitable for the implementation of his program. The MIF provides information on the structure of storage items, and on the semantics and side effects of functional operations. Familiarity with the syntax of HAL is required to understand the information provided in the MIF.

2) The Specification of the Microprogram. In this step the programmer formalizes the outline of his program. The programmer uses ABMPL to declare the abstract machine, and to specify the abstract program. The programmer must consult the WCS administrator to determine the loading address for the microcode.

3) Microprogram verification. In this step the programmer simulates the execution of the microprogram to observe its effects on the abstract machine. The abstract program shall be interpreted on a cycle-by-cycle basis. After each cycle the status of the abstract machine is displayed to observe the effects of particular microinstructions. If necessary the programmer may return to steps 1 or 2 to respecify the microprogram.

4) Microcode Generation. In this step the microprogram is compiled to generate microcode. The microcode generator optimizes the microcode. The code generated may be loaded into control storage, or saved in a library for later use.

5) Microprogram Operation. In this step the microcode is loaded, if necessary, into the Writeable Control Storage and invoked from the machine instruction level. The results of

the microprogram can now be observed under real conditions.

5.2) PRACTICAL EXAMPLES

For the purpose of this thesis I have constructed a small compiler to generate microcode for the SEL. The task of this compiler is to compose a sequential microprogram according to its ABMPL specification. The objective of this compiler is different from the HAL compiler. The ABMPL compiler seeks to generate microcode, while the HAL compiler generates information for CMPS. The ABMPL compiler depends on the Machine Information File for the production of microcode. The HAL compiler, on the other hand, generates the MIF.

The microcode generator has two main subtasks:
1) It processes the instruction set of an abstract machine and consults the machine information file (MIF) to get the microwords corresponding to each instruction. At the same time it resolves the use of literals in any instruction.
2) It processes the predicate set of an abstract machine and consults the MIF to determine the testing priority of each predicate. Each cluster in abstract program is processed to establish a sequential testing procedure and its associated actions. In this phase, merging of microwords is attempted if the sequencing operators allow it. Also the use of real addresses in control operations is resolved.

The sequential predicate testing procedures are constructed using binary decision trees so that a given test is performed no more than once.

The construction of the decision tree proceeds as follows:

1) all the information in a cluster is processed to determine the set of predicates relevant to the cluster. These predicates are then ordered in descending testing priority in a linked list.

2) a maximal decision tree is built using the relevant predicate list. The higher the priority of a test the closer it is to the root of the tree.

3) the predicates relevant to each alternative are also ordered in descending testing priority in a linked list.

4) the predicate list for each alternative is compared against the list of relevant predicates for the cluster. If within a given alternative's list an intermediate test is missing, it is inserted with an '*' qualifier to indicate that both true and false results of the test are valid and that the decision tree should reflect that fact.

5) the maximal tree is then pruned by using the predicate list of each alternative as an assertion applied to the tree. Any node of the tree not affected by any assertion is then discarded.

Each node of the tree is interpreted as a test of a condition which is either true or false. Each node has pointers associated with true and false results. The pointers may point to a further test, or in the case of leaf nodes to a list of actions within an alternative.

An example would help to illustrate. Consider the following microprogram segments:

```
          .
          .
PREDICATES
 1) ALUZ;
 2) ALUSIGN;
          .
          .
  CLUSTER 2
   1) ON 1 DO 9 RETURN.
   2) ON 2 RETURN.
   3) ON -1,-2 DO 2 GO 3.
          .
          .
```

ALUZ was declared in the MIF as "ALUZ AT +2" and ALUSIGN as "ALUSIGN AFTER +1", therefore the test ALUZ has priority over ALUSIGN. The list of relevant predicates for this cluster is ordered as follows:

1-> 2-> nil

and the maximal tree is constructed as shown in figure 5.2

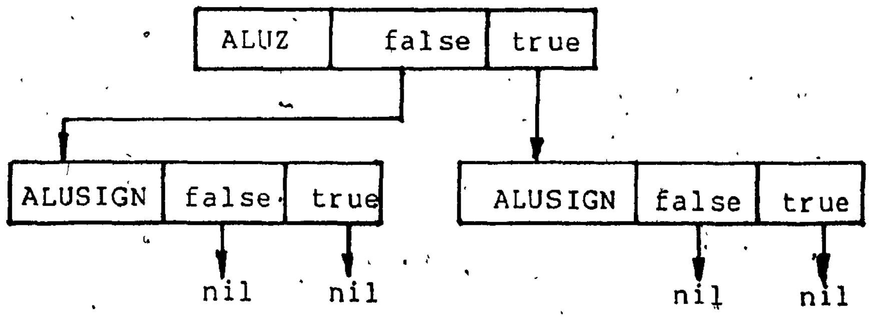


figure 5.2 Maximal Tree

for each alternative the predicates are ordered as follows:

```
1) +1-> nil
2) +2-> nil
3) -1-> -2-> nil
```

the above list are compared against the cluster's predicate list and the result is as follows:

```
1) +1-> nil
2) *1-> +2-> nil
3) -1-> -2-> nil
```

applying these assertions to the maximal tree and including the associated actions we now have the tree of figure 5.3.

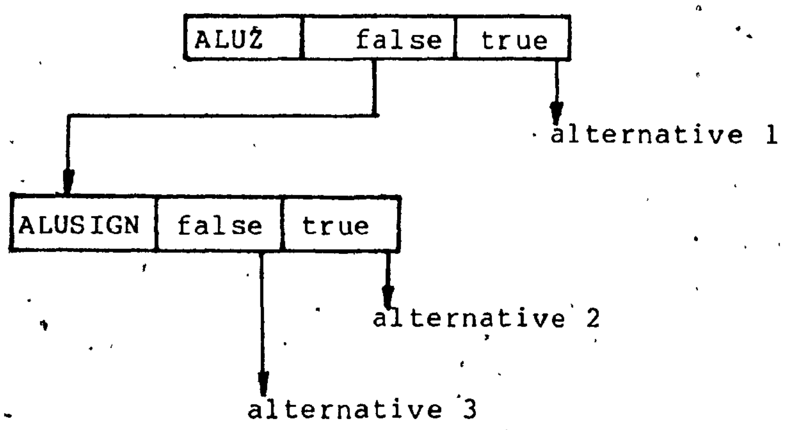


figure 5.3 Testing Tree

by traversing the tree infix left-son-first and consulting the MIF at each node, we are able to generate the following sequential code:

```
n    ON ALUZ GOTO n+4
n+1  ON ALUSIGN RETURN JSTACK
n+2  instruction 2
n+3  GOTO address cluster 3
n+4  instruction 9
n+5  RETURN JSTACK
```

Internally, the compiler structures the information of a cluster as node having an associated list of predicates, a decision tree, and a list of alternatives. Each alternative in turn is also a node, which among other things has associated a list of predicates and actions. Since the decision tree is linked with the actions of particular alternatives it is possible for the compiler to look-ahead for the purposes of optimization.

It must be pointed out that the predicates of each alternative within a cluster are expected to occur exclusively and to be non-contradictory of each other, otherwise the compiler would not generate code and the microprogrammer must respecify the cluster. In the case of conflicts in the testing priorities the compiler will attempt to resolve them, and if not possible them the microprogrammer must respecify.

A positive consequence of the structure that ABMPL imposes on a microprogram is that abstract programs can be conveniently stored. The information provided in the abstract program can provide complete look-ahead during the optimization process.

The microcode generator constructs a binary tree for each cluster in the program using the information provided in the alternatives. The leaf nodes of the tree point to a sequence of actions to be executed, including the selection of the next cluster to be processed. With each cluster represented as a binary tree with connections to other clusters, an abstract program can be considered a connected

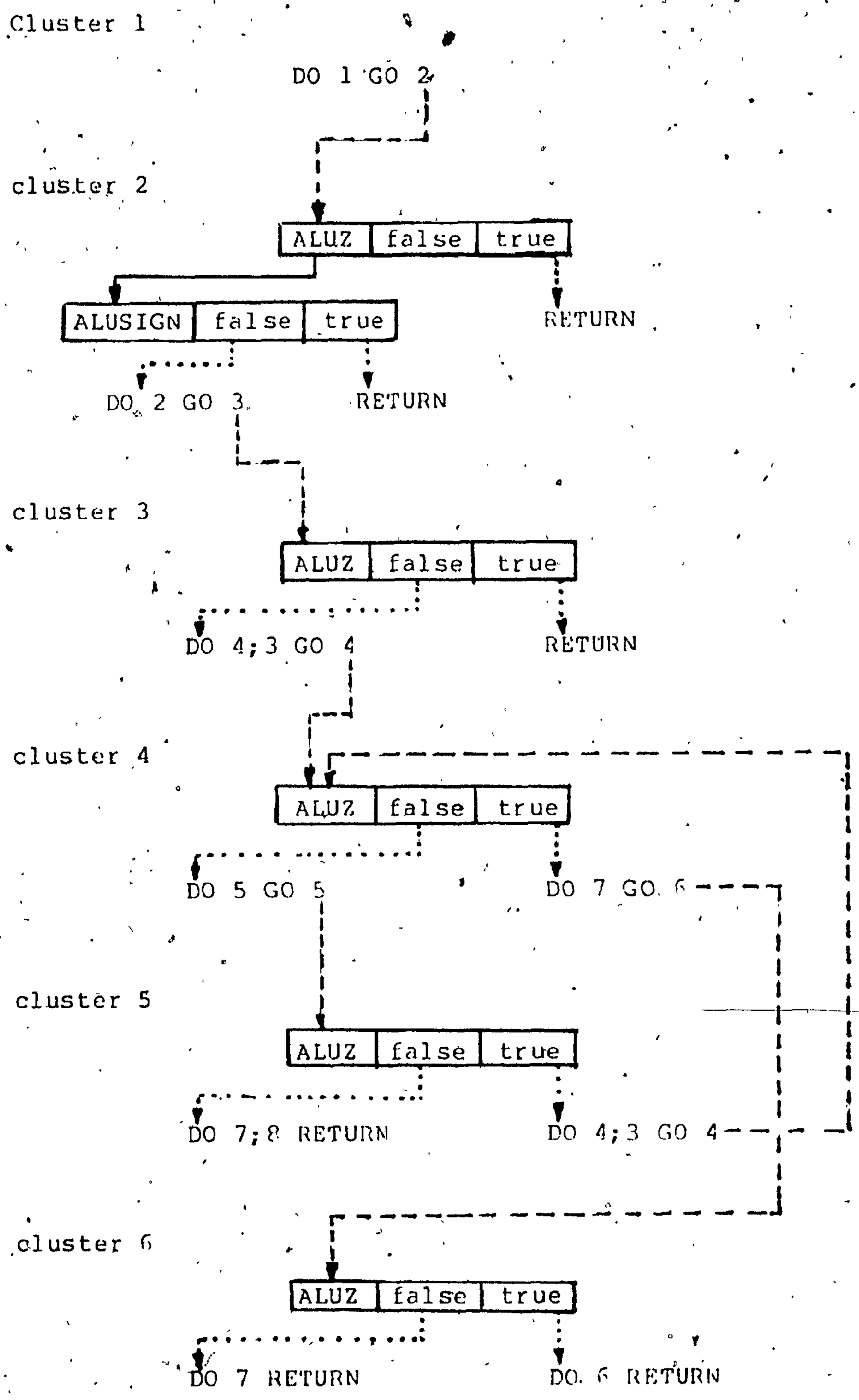


Figure 5.4 Connected Graph

graph whose nodes are binary trees [fig. 5.4].

CMPS provides local compaction through the use of sequencing operators to separate actions within an alternative. Graphs such as that of figure 5.4 could provide the basis for global optimization. In this graph there are three types of pointers, a solid line indicates a pointer to a node within the decision tree, a dotted line indicates a pointer to an alternative, and a dashed line indicates a pointer to another cluster. Dotted lines appear only in the leaf nodes of the tree. Dashed lines represent the connections between the nodes of the graph representing the microprogram.

In order to show the feasibility of CMPS two microprograms have been fully implemented for the SEL32/75. The microprograms are shown in appendix D. For these microprograms a partial description of the SEL was coded using HAL, and then it was processed by the HAL compiler to generate an MIF. The microprograms were compiled and the microcode generated was loaded and executed in the WCS of the SEL32/75. Upon execution both programs produced correct results.

As an example of the use of ABMPL I include here a microprogram designed to implement a machine instruction typical of many computers:

```
 MPROGRAM LOADREG(4200);
(*
THIS PROGRAMS EFFECTS THE MACHINE INSTRUCTION 'LOAD WORD'
IN THE SEL INSTRUCTION SET. THE FORMAT OF THIS INSTRUCTION
IS AS FOLLOWS:
              BITS Ø-5: OPCODE B'1Ø1Ø11'
              BITS 6-8: REGISTER TO BE LOADED
              BITS 9-1Ø: INDEX REGISTER
              BIT 11: INDIRECT ADDRESSING
              BITS 12-31: ADDRESS OF WORD
              ASSEMBLER CODING: LW REGISTER,WORD ADDRESS
                                                        *)
STORE MAR,            (*MEMORY ADDRESS REGISTER*)
      DI,             (*MEMORY DATA REGISTER*)
      MAIN,           (*MAIN MEMORY*)
      HIREG,          (*REG BANK SELECT*)
      IØ,             (*INSTR DECODE REG*)
      REGFILE,        (*REG FILE *)
      INDIR,          (*INDIRECTION BIT*)
      BIBUSY;         (* BUS TRANSBIT*)

ISET
 1) MAR(5..23):= SELECT(13..31,IØ+REGFILE[HIREG][IØ(9..1Ø)]) <INDIR
 2) START DI:= MAIN[MAR] <BIBUSY>;
 3) REGFILE[HIREG][IØ(6..8)]:=DI;
 4) MAR(5..23):= SELECT(13..31,DI+REGFILE[HIREG][DI(9..1Ø)]);

PREDICATES
 1) BYBUSY;
 2) INDIR;

PROGRAM
 CLUSTER 1
  1) DO 1;2 GO 2.            (*READ MAIN*)
 CLUSTER 2
  1) ON 1 GO 2.              (*READ PENDING*)
  2) ON -1,-2 DO 3 RETURN.   (*NO INDIRECTION*)
  3) ON -1,2 DO 4;2 GO 3.    (*INDIRECT READ*)
 CLUSTER 3
  1) ON 1 GO 3.              (*READ PENDING*)
  2) ON -1 DO 3 RETURN.      (*LOAD REG, QUIT*)
END.
```

The above example illustrates the implementation of one particular machine instruction. The strategy to implement the instruction set of a computer depends on how the operation codes are decoded. In the case of the SEL32/75 a Programmable Logic Array (PLA) is used to indicate the starting address of the microprogram that implements the instruction. In this case the instruction set is implemented as a series of individual, but interrelated, microprograms. Another implementation would leave the responsability of obtaining the starting addresses to the microprogrammer. In this case the instruction set would be implemented as one long microprogram with a set of entry points corresponding to the different instructions. The microprogram would have a common segment to determine the particular entry point of an instruction.

The first implementation seems to be the most common, it is used in the SEL and in the AM2900. Once the native instruction set has been microprogrammed and entry addresses selected, then the PLA would be coded accordingly.

CHAPTER 6: THE FEASIBILITY OF CMPS

The purpose of this thesis has been to design a microprogramming system that addresses the various aspects of microcode production. The end result of this research has been the design of a Comprehensive Microprogramming System (CMPS). Of special importance in the design of CMPS have been the Hardware Abstraction Language (HAL), and the Alternative Based MicroProgramming Language (ABMPL). Prototype versions of these have been implemented and tested succesfully.

Using HAL the microarchitecture can be described in a clear, and systematic manner. A clear description is the basis for an effective communication of machine information to the prospective microprogrammer. HAL defines a direct link between the functional description of operations and the microwords that implement them. It is this direct connection that allows the microcode generator to produce efficient microcode. Therefore, HAL not only describes the machine to the microprogrammer, but also to the microcode generator.

The approach taken in the design of the microprogram specification language provides multiple benefits. ABMPL imposes a well defined structure on microprograms. The microprogrammer deals with an abstract machine, which is simpler than the real machine. Local compaction of microcode is guided by the microprogrammer through the use of sequencing operators. The microprogrammer is isolated from the intricacies of the sequencing mechanism of the particular machine. The structuring of the abstract programs into clusters of alternatives provides the basis for a global optimization process.

The implementation of the various components of CMPS has been shown to be feasible. The principal merit of CMPS is not in its individual components, but rather in the fact that several apparently unrelated ideas and concepts have been brought together to deal with the problems of microcode production.

CMPS addresses the problems most commonly associated with microprogramming, namely: Machine dependence, high-level microprogramming, and microcode optimization.

6.1) MACHINE DEPENDENCE.

With respect to the problem of the machine specificity of microprograms CMPS does not attempt to produce microprograms which are universally portable, although the synonym facility of the microcode generator provides limited portability. Instead the intention has been to make the system portable, i.e., to have the ability to implement CMPS in a wide range of machines. Once the system is installed in a particular machine the microprogramming process can proceed.

The Hardware Abstraction Language (HAL) used by CMPS is flexible enough to describe a wide range of microarchitectures. Any unusual operation in a particular machine can be included by extending the language with additional predefined procedures and functions. This approach implies that the data contained in the Machine Information File (MIF) is machine specific although the language used to generate the MIF is machine independent.

No practical solution to the problem of machine dependence has been found so far. Dasgupta during the discussion of his paper on microprogramming language design [DASG80*] states that: "portability is a myth in the microprogramming context.". Microprograms are so intimately linked to the microarchitecture that the solution to this problem, if it exist, will come from future developmemts in the field of architecture design and implementation. Microprogramming software can not show generality if the architectures for which it is intended do not display the same degree of generality.

6.2) HIGH-LEVEL MICROPROGRAMMING

The main characteristic of high-level languages is the use of abstractions in programming. In the specification of microprograms a programmer working with CMPS uses abstractions produced with HAL to refer to storage items and operations. The only limitation is that the set of abstractions is finite and is determined by the characteristics of the hardware.

Allowing unlimited abstraction would add overhead to the microcode generated. Since there could be more abstract objects than hardware objects, extra code would be required to manage the binding of abstract objects to hardware objects.

Another form of abstraction offered by CMPS is found in the specification of microprograms. These are structured as a pair <abstract machine, abstract program>. The abstract machine is a subset of the real machine. This structuring allows the programmer to deal with a conceptually simpler machine.

6.3) MICROCODE OPTIMIZATION.

It is in the area of optimization where CMPS shows its greatest promise. The Alternative Based Microprogramming Language (ABMPL) provided in CMPS forces the specification of well structured programs. The information provided in an abstract program can be stored in a convenient structure to provide complete look-ahead during an optimization process.

The microcode generator can construct a graph such as that shown in figure 5.4. Such a graph provides the basis for a global optimization process. Local optimization having been achieved through the sequencing operators of

ABMPL. The objective of an optimization process should be to minimize the number of nodes in the graph. This could be achieved by attempting to combine some of the trees in the graph. This process could draw from the ample experience there is in the manipulation of trees.

6.4) CONCLUSIONS

During the course of this research the most important lesson that has been learned is that the production of good microcode is not a one-dimensional problem. Microcode production is the ultimate objective in a hierarchy of interrelated problems, such as microarchitecture description, high-level microprogramming, microprogram correctness, and microcode optimization.

In contemporary microprogramming research too much emphasis has been placed on the introduction of high-level languages to generate microcode. High-level microprogramming should be a consequence, not the major objective, of a comprehensive solution to the production of microcode.

At the same time little emphasis has been placed in communicating architecture information to the microprogrammer. Once it is possible to provide a programmer with a grasp of the limits and abilities of a machine, then attention can be focused on the tools that will allow him to microprogram. This conclusion is supported by Richter, who in the conclusion of his paper on microcode generation [RICH8Ø] states: "the research for the development of complex language tools for microprogramming has to concetrate on the well-structured modelling of the description of the resources and primitives at the lower level of potential target architectures".

A solution to the problems of microcode production must be comprehensive in the sense that it addresses the various aspects of the microprogramming process. The microprogramming system proposed in this thesis has attempted to do just that, by designing a set of interrelated and mutually complementing components. Richter supports this approach when he states [RICH8Ø]: "finally it seems necessary to consider microcode generation, optimization, and testing as an integral task which has to be supported with identical and mutually completing tools".

The most difficult problem encountered in the preparation of this thesis was the production of a structured description of the microachitecture. The hardware of computers often includes ad hoc features such as specific purpose data paths and hardware assists (ex: floating point assist, page map assist). These features are designed to enhance the performance of the machine, but they tend to be detrimental to the structure of the architecture.

The degree of structure in the hardware should improve as the use of structured design notations (such as ISPS) becomes widespread, and components are standardized. This will facilitate the description of microarchitectures and should have a positive effect on microprogramming software.

The Hardware Abstraction Language included in CMPS could be used to design a computer. In this case a HAL description of the machine would be coded according to the architectural goals. Later on, the necessary hardware would be assembled to satisfy the HAL specifications. This would be the reverse of the process we have followed in this thesis. Using AM2900's components to build the target architecture such an approach could become a reality.

REFERENCES

AMD1 Advanced Microdevices, Am2900 Learning and Evaluation Kit User's Manual.

BABA81 The MPG System: A Machine Independent Efficient Microprogram Generator, T. Baba & H. Hagiwara, IEEE Transactions on Computers, June 1981.

BAER80 Computer Systems Architecture, Jean-Loup Baer, Computer Science Press, The control Unit and Microprogramming, p. 328

BARB81 Instruction Set Processor Specification (ISPS): The Notation and its Application, M. Barbacci, IEEE Transactions on Computers, January 1981

BELL71 Computers Structures: Readings and Examples, C. G. Bell & A. Newell, the ISP notation, Mcgraw-Hill 1971.

BELL82 Computers Structures: Principles and Examples, Siewiorek/Bell/Newell, chapter 13, Mcgraw-Hill 1982.

BERG80 Firmware Engineering: Remarks and Strategy, H.K. Berg & W.R. Franta, IFIP Conference on Firmware, Microprogramming and Restructurable Hardware, North-Holland Editors 1980, QA 76.6 I189.

CHAP75 V. Chaptal, Problems of Microprogram Production, Infotec State of the Art Report on Microprogramming and Systems Architecture, report #23, 1975, QA 76.6 M5

DASG80 Computing Surveys, Sept. 80, Some Aspects of High Level Microprogramming, S. Dasgupta

DASG80* Microprogramming Language Design, S. Dasgupta, IFIP Conf. On Firmware, Microprogramming & Restructurable Hardware, May 1980, North Holland, QA 76 I189.

DASG79 Computing Surveys, march 1979, The Organization of Microprogram Stores, S. Dasgupta

DAVI80 Firmware Engineering: An Extensive Update, S. Davidson & B. Shriver, IFIP Conference on Firmware, Microprograming and Restructurable Hardware, North-Holland 1980, QA 76.6 I189.

DEWI76 Extensibility: A New Approach for Designing Microprogramming Languages, D. Dewitt, Proceedings 9th Microprogramming Workshop, SIGMICRO September

1976

FREE75	Software Systems Principles, Microprogramming, P. Freeman, SRA 1975.
FLYN8Ø	Interpretation, Microprogramming, and the Control of a Computer, M.J. Flynn, Introduction to Computer Architecture, SRA 198Ø, QA76.9.A73I57.
GILO8Ø	FIT - Firmware Specification, Implementation, Validation, W. K. Giloi, P. Behr & R. Gueth, Firmware, Microprogramming and Restructurable Hardware, Proccedings IFIP Working Conference, North Holland, QA 76.6 I189 198Ø.
GUHA77	Dynamic Microprograming in a Time Sharing Enviroment, R. K. Guha, Proceedings 1Øth Micro Workshop, SIGMICRO 1977.
HEIN8Ø	A Data Abstraction Language Based on Microprogramming, R. Kurki-suonio & J. Heinanen, Proceedings 13th Microprogramming Workshop, SIGMICRO 198Ø
JAWO81	Controlled Program Design by Use of the ABL Programming Concept, W.M. Jaworski & H. Hinterberger, Applied Informatics July 1981.
JONE75	Instruction Sequencing in Microprogrammed Computers, Louise H. Jones, Microprogramming by Sondak & Mallach, QA 76.6 M48
LAND8Ø	Local Microcode Compaction Techniques, D. Landskov, S. Davidson, B. Shriver & P. Mallet, Computing Surveys September 198Ø
MALI78	Design Objectives for High-level Microprogramming Languages, K. Malik & T. Lewis, 11th Micro Workshop, SIGMICRO December 78.
MICK77	SIG Micro, june 77, Microprogramming for the Hardware Enginneer, John R. MICK
PATT76	Strum: Structured Microprogram Development System for Correct Firmware, D. Patterson, IEEE Transactions on Computers October 1976
PERS77	Design of a Microprogram Generator for the Varian V73, M. Persson, SIG Micro, december 1977
RAMA74	A High Level Language for Horizontal Microprogramming, C. Ramamoorthy & M. Tsuchiya, IEEE Transactions on Computers August 1974

RAUS80 Microprogramming: a Tutorial and Survey of Recent Developments, T. G. Rauscher & P. M. Adams, IEEE transactions on computers, january 1980

RICH80 Extensions for Microcode Generation & Verification, L. Richter, IFIP Conf. On Firmware, Microprogramming & Restructurable Hardware, North Holland May 1980, QA 76 I189.

SEL1 SEL 32/75 WCS Reference Manual

SEL2 SEL 32/75 Microassembler Technical Manual

SHRI81 IEEE Transactions on Computers, July 81, Introduction, T. Lewis & B. Shriver

SINT80 A Survey of High Level Microprogramming Languages, Marleen Sint, Proceedings 13th Anual Microprogramming Workshop, SIGMICRO December 1980

SINT81 MIDL: a Microinstruction Description Language, M. Sint, Proceeedings 14th Microprogramming Workshop, SIGMICRO Decemcber 81.

STEV64 Bell & Newell Computer Structures, The Structure of System 360 Part II, P. 604, W. Y. Stevens

TUCK65 Communications ACM, december 65, Emulations of Large Systems, S. G. Tucker

WILK69 Computing Surveys, Sept. 69, The Growth of Interest in Microprogramming: A Literature Survey, M. V. Wilkes

WIRT73 N. Wirth, Systematic Programming: An Introduction, Procedures & Functions, p. 93, Prentice-Hall 1973

APPENDIX A - THE GRAMMAR OF ABMPL

the grammar of ABMPL follows:

```
 <microprogram> ::=
  MPROGRAM <identifier>(<base address><parm>);
   <machine>
   <program>
  END.

 <parm> ::= ,<storage items> !
            <empty>

 <machine> ::= <storage>
               <synonyms>
               <microprocedures>
               <instruction set>
               <predicates>

 <storage> ::= STORE <storage items>;

 <storage items> ::= <valid identifier> !
                    <valid identifier>, <storage items>

 <synomyms> ::= SYN <synonym list>

 <synonym list> ::= <synonym>; !
                    <synonym>; <synonym list>

 <synonym> ::= <valid identifier> = <identifier>

 <microprocedures> ::= <microprocedure> <microprocedure
                       <empty>

 <microprocedure> ::= MPROCEDURE <identifier><parameters>;
                       <microprocedure block>
                      END;

 <parameters> ::= (<storage items>) !
                  <empty>

 <microprocedure block> ::= <machine> <procedure> !
                            EXT <base address>;

 <procedure> ::= PROCEDURE <cluster list>

 <instruction set> ::= ISET <instruction list> !
                       <empty>

 <instruction list> ::= <instruction>; !
                        <instruction>; <instruction list>

 <instruction> ::=
 <i code>) <valid operation> <implicit assigments>

<implicit assignment> ::=  '<'<storage items>'>' !
                          <empty>

<predicates> ::= PREDICATES <predicate list> !
                 <empty>

<predicate list> ::= <predicate>; !
                     <predicate>; <predicate list>

<predicate> ::= <p code>) <expression>

<program> ::= PROGRAM <cluster list>

<cluster list> ::= <cluster> !
                   <cluster> <cluster list>

<cluster> ::= CLUSTER <c code> <alternative list>

<alternative list> ::= <alternative>. !
                       <alternative>. <alternative list>

<alternative> ::= <alternative code>) <abs statement>

<abs statement> ::=
 <abs condition> <abs actions> <abs control>

<abs condition> ::= ON <predicate references> !
                    <empty>

<abs actions> ::= DO <instruction sequence> !
                  <empty>

<abs control> ::= GO <c code> !
                  RETURN

<predicate references> ::=
 <predicate reference> !
 <predicate reference>, <predicate references>

<predicate reference> ::= <sign> <p code>

<instruction sequence> ::=
 <i code> !
 <i code><seq op><instruction sequence>

<seq op> ::= , ! ;

<sign> ::= + ! - ! <empty>

<i code> ::= <characters>

<p code> ::= <characters>
```

```
<c code> ::= <characters>

<alternative code> ::= <characters>

<identifier> ::= <letter> <characters>

<characters> ::= <character><characters> !
                 <character>

<character> ::= <letter> !
                <digit>

<digit> ::= Ø ! 1 ! 2 ! 3 ! 4 !
            5 ! 6 ! 7 ! 8 ! 9

<letter> ::= A ! B ! C ! D ! E ! F ! G ! H ! I !
             J ! K ! L ! M ! N ! O ! P ! Q ! R !
             S ! T ! U ! V ! W ! X ! Y ! Z
```

APPENDIX B - THE GRAMMAR OF HAL

The grammar of the Hardware Abstraction Language follows:

```
<machine> ::= MACHINE <identifier>;
              <machine description>
              END.

<machine description> ::= <data structure>
                          <functional operations>
                          <control structure>

<data structure> ::= <data types> <storage>

<data types> ::= TYPE <types> !
                 <empty>

<types> ::= <type declaration>; !
           <type declaration>; <types>

<type declaration> ::= <identifier>= <type>

<type> ::= STACK[<positive constant>] OF <type> !
          SEQ (<constant>..<constant>) OF <bit type> !
          ARRAY [<index type>] OF <type> !
          TUPLE <field list> END !
          <simple type>

<bit type> ::= BIT ! ZERO ! ONE

<index type> ::= <simple type> !
                 <simple type>,<index type>

<field list> ::= <identifiers declaration> !
                 <identifiers declaration>;<field list>

<identifiers declaration> ::= <identifier list>: <type>

<identifier list> ::= <identifier> !
                      <identifier>,<identifier list>

<simple type> ::= <identifier> !
                  <bit type> !
                  <constant>..<constant>

<storage> ::= STORE <storage items>

<storage items> ::=
 <identifiers declaration>; !
 <identifiers declaration>; <storage items>
```

```
<functional operations> ::= OPERATION <operation list>

<operation list> ::= <operation> !
                     <operation>; <operation list>

<operation> ::=
 EXP: <explicit operations> <underlying operations>

<explicit operations> ::= <statement> !
                          <compound statement>

<compound statement> ::= COCYCLE <statement list> END

<statement list> ::=
 <statement> !
 <statement><sequence operator> <statement list>

<sequence operator> ::= , ! ;

<underlying operations> ::=
 IMP: <implicit operations> MIW: <microword> !
 MIW: <microword>

<implicit operations> ::=
 <implicit operation> !
 <implicit operation>; <implicit operations>

<implicit operation> ::= <statement> !
                         <compound statement>

<microword> ::= '<bit string>'

<bit string> ::= <bit description> !
                <bit description><bit string>

<bit description> ::= Ø ! 1 ! - ! <letter>

<statement> ::= <simple statement> !
               <conditional statement>

<simple statement> ::=
 START <variable concatenation>:= <expression> !
 <variable concatenation>:= <expression> !
 NULL:= <expression> !
 <predefined procedure call>

<conditional statement> ::= ON <expression> DO (<actions>)

<actions> ::= <simple statement> !
             <simple statement>,<actions>

<variable concatenation> ::=
 <variable reference> !
 <variable reference>&<variable concatenation>

<variable reference> ::= <variable> !
                         <variable>(<bit index>)

<bit index> ::= <expression> !
              <expression>..<expression>

<variable> ::= <identifier><specifiers>

<specifiers> ::= <specifier><specifiers> !
                <empty>

<specifier> ::= .<identifier> !
              [<expression list>]

<expression list> ::= <expression> !
                     <expression>,<expression list>
<expression> ::=
 <simple expression> !
 <simple expression> <relational op> <simple expression>

<relational op> ::= # ! = ! < ! > ! <= ! >=

<simple expression> ::=
 <sign><term> !
 <sign><term> <add op> <simpler expression>

<simpler expression> ::=
 <term> !
 <term> <add op> <simpler expression>

<sign> ::= + ! -

<add op> ::= OR ! XOR ! + ! -

<term> ::= <factor> !
        <factor> <multiply op> <term>

<multiply op> ::= AND ! * ! /

<factor> ::= <concatenation> !
            <predefined function reference> !
            (<expression>) !
            NOT <factor>

<concatenation> ::= <element> !
                  <element>&<concatenation>

<element> ::= <positive constant> !
            <variable reference> !
            <literal indicator>

<literal indicator> ::= $<letter>
```

```
<control structure> ::=
 <tests description> <control operations>

<tests description> ::= TEST <test list>

<test list> ::= <test>; !
              <test>; <test list>

<test> ::= <expression> <timing operator> <time>

<timing operator> ::= AT ! AFTER

<time> ::= <sign> <phase>

<phase> ::= <positive integer> !
           <positive integer>.<phase>

<control operations> ::= CONTROL <control list>

<control list> ::= <control operation>; !
                  <control operation>; <control list>

<control operation> ::=
 EXP: <control description> <underlying operations>

<control description> ::=
 <condition> <control operator> <concatenation>

<condition> ::= ON <expression> !
              <empty>

<control operator> ::= GOTO ! GOSUB ! RETURN

<predefined procedure call> ::=
 SHIFTC(<shift parameters>) !
 PUSH(<storage id>,<stack id>) !
 POP(<stack id>,<storage id>)

<stack id> ::= <variable concatenation>

<storage id> ::= <variable concatenation>

<predefined function reference> ::=
 SHIFTV(<shift parameters>) !
 SELECT(<bit index>,<expression>)

<shift parameters> ::=
 <direction>,<quantity>,<fill bits>,<shift item>

<direction> ::= RIGHT ! LEFT

<quantity> ::= <concatenation>
```

```
<fill bits> ::= <concatenation>

<shift item> ::= <variable concatenation>

<identifier> ::= <letter><characters>

<characters> ::= <character><characters> !
                 <empty>

<character> ::= <letter> ! <digit>

<constant> ::= <sign><positive constant>

<positive constant> ::= <decimal> !
                        X'<hex number>' !
                        O'<octal number>' !
                        B'<binary number>'

<decimal> ::= <digit> !
              <digit><decimal>

<digit> ::= Ø ! 1 ! 2 ! 3 ! 4 !
        5 ! 6 ! 7 ! 8 ! 9

<hex number> ::= <hex digit> !
                 <hex digit><hex number>

<hex digit> ::= <digit> !
            A ! B ! C ! D ! E ! F

<octal number> ::= <octal digit> !
                   <octal digit><octal number>

<octal digit> ::= Ø ! 1 ! 2 ! 3 ! 4 ! 5 ! 6 ! 7

<binary number> ::= <binary digit> !
                    <binary digit><binary number>

<binary digit> ::= Ø ! 1

<letter> ::= A ! B ! C ! D ! E ! G ! H ! I !
         J ! K ! L ! M ! N ! O ! P ! Q !
         R ! S ! T ! U ! V ! W ! X ! Y ! Z
```

APPENDIX C - PARTIAL DESCRIPTION OF SEL 32/75

The program that processes the HAL description of the microarchitecture is called the Machine Information File Generator. This program was constructed using the recursive descend method of compilation.

For the purpose of this thesis the MIF generator is exclusively concerned with the generation of the information required to produce microcode through ABMPL. The descriptions of functional operations, tests, and control operations are parsed to generate records.

For both functional and control operations, the characters in an explicit operation are hashed to produce a numeric code to be associated with its corresponding microword. Literal information is also included in the record.

The characters in a test description are also hashed to produce a numeric code associated with a priority number. This priority is determined from the keywords 'AT' and 'AFTER', with 'AT' tests having the highest priority.

The output of the HAL compiler is shown in the following pages.

```
CONCORDIA UNIVERSITY COMPREHENSIVE MICROPROGRAMMING SYSTEM:  MACHINE INFORMATION FILE GENERATOR  VER
 HOST: CYBER 170 NOS 1.3  TARGET: SEL32/75   82/07/29.   09.40.46.

MACHINE SEL3275;

 TYPE WORD= SEQ (0..31) OF BIT;
      MWORD= SEQ (0..47) OF BIT;
      REGISTER= SEQ (0..31) OF BIT;
      REGBANK= ARRAY [0..15] OF REGISTER;

 STORE S,T,IO,I1,DI: REGISTER;
       LIT,NCTR: SEQ (0..7) OF BIT;
       MAR: SEQ (0..23) OF BIT;
       PC: SEQ (0..21) OF BIT;
       MPC: SEQ (0..12) OF BIT;

       SCRATCH: ARRAY [0..255] OF WORD;

       REGFILE: ARRAY [0..1] OF REGBANK;

       MAIN: ARRAY [0..262143] OF WORD;

       CSTORAGE: TUPLE
                  CROM: ARRAY [0..4095] OF MWORD;
                  WCS : ARRAY [4096..6143] OF MWORD
                 END;

       JSTACK: STACK[4] OF SEQ (0..12) OF BIT;

       ALUZ,NALUZ,ALUNEG,ALU47Z: BIT;
       ALUSIGN: BIT;
       BIBUSY: BIT;
       HIREG: BIT;

 OPERATION

  EXP: COCYCLE
       DI:= REGFILE[HIREG][$R],
       ALUSIGN:= REGFILE[HIREG][$R](0)
      END
  IMP: COCYCLE
       ON REGFILE[HIREG][$R] = 0 DO (ALUZ:= 1, NALUZ:= 0, ALUNEG:= 0),
       ON REGFILE[HIREG][$R] < 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 1),
       ON REGFILE[HIREG][$R] > 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 0)
      END
  MIW: '1111---------100000001100010010----RRRR------------------------';

  EXP: DI:= REGFILE[HIREG][$R]
  IMP: COCYCLE
       ON REGFILE[HIREG][$R] = 0 DO (ALUZ:= 1, NALUZ:= 0, ALUNEG:= 0),
       ON REGFILE[HIREG][$R] < 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 1),
       ON REGFILE[HIREG][$R] > 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 0)
      END
  MIW: '1111---------1000000110000---------RRRR------------------------';
```

```
CONCORDIA UNIVERSITY COMPREHENSIVE MICROPROGRAMMING SYSTEM:  MACHINE INFORMATION FILE GENERATOR  VER
 HOST: CYBER 170 NOS 1.3  TARGET: SEL32/75   82/07/29.   09.40.47.

 EXP:  NULL:= S
 IMP:  COCYCLE
       ON S = O DO (ALUZ:= 1, NALUZ:= 0, ALUNEG:= 0),
       ON S < O DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 1),
       ON S > O DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 0)
      END
 MIW:  '----------000---11100000-----------------------------------------------';

 EXP:  COCYCLE
       LIT:=$L,
       NULL:= X'000000'&LIT AND DI
      END
 IMP:  COCYCLE
       ON X'000000'&LIT AND DI = 0 DO (ALUZ:= 1, NALUZ:= 0, ALUNEG:= 0),
       ON X'000000'&LIT AND DI < 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 1),
       ON X'000000'&LIT AND DI > 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 0)
      END
 MIW:  '----------01101111010000---00010----LLLLLLLL--------------------';

 EXP:  COCYCLE
       LIT:= $L,
       NCTR:= SELECT(0..7,LIT&X'000000')
      END
 IMP:  COCYCLE
       ON LIT&X'000000' = 0 DO (ALUZ:= 1, NALUZ:= 0, ALUNEG:= 0),
       ON LIT&X'000000' > 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 0),
       ON LIT&X'000000' < 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 1)
      END
 MIW:  '----------000---11101100---00010----LLLLLLLL--------------------';

 EXP:  NCTR:= NCTR - 1
 MIW:  '0000---011-----------------------------------------------------------';

 EXP:  T:= DI
 IMP:  COCYCLE
       ON DI = 0 DO (ALUZ:= 1, NALUZ:= 0, ALUNEG:= 0),
       ON DI < 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 1),
       ON DI > 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 0)
      END
 MIW:  '-------------01100001111------------------------------------------';

 EXP:  T:= REGFILE[HIREG][$R] - 1
 IMP:  COCYCLE
       ON REGFILE[HIREG][$R] - 1 = 0 DO (ALUZ:= 1, NALUZ:= 0, ALUNEG:= 0),
       ON REGFILE[HIREG][$R] - 1 < 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 1),
       ON REGFILE[HIREG][$R] - 1 > 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 0)
      END
 MIW:  '0000-----100---01001111000---------RRRR-------------------------';

 EXP:  COCYCLE
```

```
CONCORDIA UNIVERSITY COMPREHENSIVE MICROPROGRAMMING SYSTEM:   MACHINE INFORMATION FILE GENERATOR . VERS
 HOST: CYBER 170 NOS 1.3  TARGET: SEL32/75   82/07/29.   09.40.47.

        LIT:= $L,
        T:= X'FFFFFF'&LIT + T
       END
 IMP:  COCYCLE
        ON X'FFFFFF'&LIT + T = 0 DO (ALUZ:= 1, NALUZ:= 0, ALUNEG:= 0),
        ON X'FFFFFF'&LIT + T < 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 1),
        ON X'FFFFFF'&LIT + T > 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 0)
       END
 MIW:  '----------11100000111111-----------LLLLLLLL--------------------';

 EXP:  S:= 0
 MIW:  '0000000-----------------------------------0111-----------------';

 EXP:  S:= S + T
 IMP:  COCYCLE
        ON S + T = 0 DO (ALUZ:= 1, NALUZ:= 0, ALUNEG:= 0),
        ON S + T < 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 1),
        ON S + T > 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 0)
       END
 MIW:  '----------00000000110001---------------------------------------';
C-4
 EXP:  S:= S - T
 IMP:  COCYCLE
        ON S - T = 0 DO (ALUZ:= 1, NALUZ:= 0, ALUNEG:= 0),
        ON S - T < 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 1),
        ON S - T > 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 0)
       END
 MIW:  '----------00000001010001---------------------------------------';

 EXP:  SHIFTC(RIGHT,1,0,S&DI)
 MIW:  '0000--------------------000010011----1000----------------------';

 EXP:  SHIFTC(RIGHT,1,S(0),S&DI)
 MIW:  '0000--------------------000010011----1001----------------------';

 EXP:  REGFILE[HIREG][$R]:= DI
 IMP:  COCYCLE
        T:= DI,
        ON DI = 0 DO (ALUZ:= 1, NALUZ:= 0, ALUNEG:= 0),
        ON DI < 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 1),
        ON DI > 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 0)
       END;
       REGFILE[HIREG][$R]:= T
 MIW:  '-------------01100001000-----------RRRR------------------------';

 EXP:  REGFILE[HIREG][$R]:= REGFILE[HIREG][$R] + 1
 IMP:  COCYCLE
        T:= REGFILE[HIREG][$R] + 1,
        ON REGFILE[HIREG][$R] + 1 = 0 +> (ALUZ:= 1, NALUZ:= 0, ALUNEG:= 0),
        ON REGFILE[HIREG][$R] + 1 < 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 1),
        ON REGFILE[HIREG][$R] + 1 > 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 0)
```

```
      END;
      REGFILE[HIREG][$R]:= T
 MIW: '0000------100---0110100000---------RRRR-----------------------';

 EXP: REGFILE[HIREG][$R]:= S
 IMP: COCYCLE
       T:= S,
       ON S = 0 DO (ALUZ:= 1, NALUZ:= 0, ALUNEG:= 0),
       ON S < 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 1),
       ON S > 0 DO (ALUZ:= 0, NALUZ:= 1, ALUNEG:= 0)
      END;
      REGFILE[HIREG][$R]:= T
 MIW: '----------000---11101000------------RRRR-----------------------';

TEST ALUSIGN AFTER +1;
     ALUZ AT +2;
     NALUZ AT +2;
     NCTR = 0 AFTER +1;

CONTROL
 EXP: GOTO $A
 IMP: COCYCLE
        MPC:= MPC +1,
        MPC(9..12):= $A
       END
 MIW: '0000100-----------------------------------AAAA---------------';

 EXP: GOTO $A
 IMP: COCYCLE
        MPC:= MPC +1,
        MPC(5..12):= $A
       END
 MIW: '0000101-------------------------------AAAAAAAA---------------';

 EXP: GOTO $A
 IMP: COCYCLE
        MPC:= MPC +1,
        MPC(1..12):= $A
       END
 MIW: '0000110---------------------------AAAAAAAAAAAA---------------';

 EXP: GOTO $A
 IMP: COCYCLE
        MPC:= MPC +1,
        MPC:= $A
       END
 MIW: '0000111---------------------------AAAAAAAAAAAA---------------';

 EXP: ON ALUZ GOTO $A
 IMP: COCYCLE
```

```
       ON ALUZ DO (MPC:=MPC+1,MPC(9..12):= $A),
       ON NOT ALUZ DO (MPC:= MPC + 1)
     END
 MIW: '1011100----------------------------------AAAA---------------';

 EXP: ON ALUZ GOTO $A
 IMP: COCYCLE
       ON ALUZ DO (MPC:=MPC+1,MPC(5..12):= $A),
       ON NOT ALUZ DO (MPC:= MPC + 1)
     END
 MIW: '1011101-------------------------------AAAAAAA---------------';

 EXP: ON ALUZ GOTO $A
 IMP: COCYCLE
       ON ALUZ DO (MPC:=MPC+1,MPC(1..12):= $A),
       ON NOT ALUZ DO (MPC:= MPC + 1)
     END
 MIW: '1011110--------------------------AAAAAAAAAAAA---------------';

 EXP: ON ALUZ GOTO $A
 IMP: COCYCLE
       ON ALUZ DO (MPC:= $A),
       ON NOT ALUZ DO (MPC:= MPC + 1)
     END
 MIW: '1011111--------------------------AAAAAAAAAAAA---------------';

 EXP: ON ALUSIGN RETURN JSTACK
 IMP: COCYCLE
       ON ALUSIGN DO (POP(JSTACK,MPC)),
       ON NOT ALUSIGN DO (MPC:= MPC + 1)
     END
 MIW: '0011001----------------------0001---------------------------';

 EXP: ON ALUSIGN GOTO $A
 IMP: COCYCLE
       ON ALUSIGN DO (MPC:=MPC+1,MPC(9..12):= $A),
       ON NOT ALUSIGN DO (MPC:= MPC + 1)
     END
 MIW: '0011100----------------------0001--------AAAA---------------';

 EXP: ON ALUSIGN GOTO $A
 IMP: COCYCLE
       ON ALUSIGN DO (MPC:=MPC+1,MPC(5..12):= $A),
       ON NOT ALUSIGN DO (MPC:= MPC + 1)
     END
 MIW: '0011101----------------------0001-----AAAAAAA---------------';

 EXP: ON ALUSIGN GOTO $A
 IMP: COCYCLE
       ON ALUSIGN DO (MPC:=MPC+1,MPC(1..12):= $A),
       ON NOT ALUSIGN DO (MPC:= MPC + 1)
```

```
CONCORDIA UNIVERSITY COMPREHENSIVE MICROPROGRAMMING SYSTEM:  MACHINE INFORMATION FILE GENERATOR  VERS
 HOST: CYBER 170 NOS 1.3  TARGET: SEL32/75   82/07/29.   09.40.47.

        END
    MIW:  '0011110--------------------------0001AAAAAAAAAAAA-----------------';

    EXP:  ON ALUSIGN GOTO $A
    IMP:  COCYCLE
          ON ALUSIGN DO (MPC:= $A),
          ON NOT ALUSIGN DO (MPC:= MPC + 1)
        END
    MIW:  '0011111--------------------------000AAAAAAAAAAAAA-----------------';

    EXP:  ON NALUZ GOTO $A
    IMP:  COCYCLE
          ON NALUZ DO (MPC:=MPC+1,MPC(9..12):= $A),
          ON NOT NALUZ DO (MPC:= MPC + 1)
        END
    MIW:  '1100100--------------------------------------AAAA-----------------';

    EXP:  ON NALUZ GOTO $A
    IMP:  COCYCLE
          ON NALUZ DO (MPC:=MPC+1,MPC(5..12):= $A),
          ON NOT NALUZ DO (MPC:= MPC + 1)
        END
    MIW:  '1100101----------------------------------AAAAAAAA-----------------';

    EXP:  ON NALUZ GOTO $A
    IMP:  COCYCLE
          ON NALUZ DO (MPC:=MPC+1,MPC(1..12):= $A),
          ON NOT NALUZ DO (MPC:= MPC + 1)
        END
    MIW:  '1100110------------------------------AAAAAAAAAAAA-----------------';

    EXP:  ON NALUZ GOTO $A
    IMP:  COCYCLE
          ON NALUZ DO (MPC:= $A),
          ON NOT NALUZ DO (MPC:= MPC + 1)
        END
    MIW:  '1100111-----------------------------AAAAAAAAAAAAA-----------------';

    EXP:  ON NCTR = 0 GOTO $A
    IMP:  COCYCLE
          ON NCTR = 0 DO (MPC:=MPC+1,MPC(9..12):= $A),
          ON NCTR # 0 DO (MPC:= MPC + 1)
        END
    MIW:  '0010100------------------------------1001----AAAA-----------------';

    EXP:  ON NCTR = 0 GOTO $A
    IMP:  COCYCLE
          ON NCTR = 0 DO (MPC:=MPC+1,MPC(5..12):= $A),
          ON NCTR # 0 DO (MPC:= MPC + 1)
        END
    MIW:  '0010101------------------------------1001AAAAAAAA-----------------';
```

```
CONCORDIA UNIVERSITY COMPREHENSIVE MICROPROGRAMMING SYSTEM:  MACHINE INFORMATION FILE GENERATOR  VER
 HOST: CYBER 170 NOS 1.3  TARGET: SEL32/75   82/07/29.   09.40.48.

  EXP: ON NCTR = 0 RETURN JSTACK
  IMP: COCYCLE
       ON NCTR = 0 DO (POP(JSTACK,MPC)),
       ON NCTR # 0 DO (MPC:= MPC + 1)
      END
  MIW: '0010001-----------------------------1001------------------------';

  EXP: ON ALUZ GOSUB $A
  IMP: COCYCLE
       ON ALUZ DO (MPC:=MPC+1,PUSH(MPC,JSTACK), MPC:= $A),
       ON NOT ALUZ DO (MPC:= MPC + 1)
      END
  MIW: '1011111111-------------------------AAAAAAAAAAAAA----------------';

  EXP: GOSUB $A
  IMP: COCYCLE
        MCP:= MPC + 1,
       PUSH(MPC, JSTACK),
       MPC(9..12):= $A
      END
  MIW: '0000100111----------------------------------AAAA----------------';

  EXP: GOSUB $A
  IMP: COCYCLE
        MCP:= MPC + 1,
       PUSH(MPC,JSTACK),
       MPC(5..12):= $A
      END
  MIW: '0000101111------------------------------AAAAAAAA----------------';

  EXP: GOSUB $A
  IMP: COCYCLE
        MCP:= MPC + 1,
       PUSH(MPC,JSTACK),
       MPC(1..12):= $A
      END
  MIW: '0000110111--------------------------AAAAAAAAAAAA----------------';

  EXP: GOSUB $A
  IMP: COCYCLE
        MCP:= MPC + 1,
       PUSH(MPC,JSTACK),
       MPC:= $A
      END
  MIW: '0000111111-------------------------AAAAAAAAAAAAA----------------';

  EXP: ON ALUZ RETURN JSTACK
  IMP: COCYCLE
       ON ALUZ DO (POP(JSTACK, MPC)),
       ON NOT ALUZ DO (MPC:= MPC + 1)
```

```
CONCORDIA UNIVERSITY COMPREHENSIVE MICROPROGRAMMING SYSTEM:  MACHINE INFORMATION FILE GENERATOR  VERSI
 HOST: CYBER 170 NOS 1.3  TARGET: SEL32/75   82/07/29.   09.40.48.

      END
  MIW: '1011001---------------------------------------------------------------';

  EXP: RETURN JSTACK
  IMP: POP(JSTACK, MPC)
  MIW: '0000001---------------------------------------------------------------';

END.
```

MACHINE INFORMATION FILE FOLLOWS:

```
OP 240    0 0   0 0 -------------01100001111----------------------------------------
OP 348    0 0   0 0 ----------000---11100000----------------------------------------
OP 532    0 0   0 0 ----------00000000110001----------------------------------------
OP 537    0 0   0 0 ----------00000001010001----------------------------------------
OP 2109 R 36 4   0 0 ----------000---11101000------------RRRR------------------------
OP 2139 R 36 4   0 0 -------------01100001000------------RRRR------------------------
OP 3351 R 36 4   0 0 1111---------10000000110000---------RRRR------------------------
OP 6091    0 0   0 0 0000000-------------------------------------0111----------------
OP 7343 L 36 8   0 0 ----------11100000111111------------LLLLLLLL--------------------
OP 10052    0 0   0 0 0000---011------------------------------------------------------
OP 15193 L 36 8   0 0 ----------01101111010000---00010----LLLLLLLL--------------------
OP 21018    0 0   0 0 0000-----------------------000010011----1000--------------------
OP 21707 R 36 4   0 0 0000------100---01001111000---------RRRR------------------------
OP 25421    0 0   0 0 0000-----------------------000010011----1001--------------------
OP 36003 R 36 4   0 0 0000------100---01101000000---------RRRR------------------------
OP 58684 R 36 4   0 0 1111---------1000000011000010010----RRRR------------------------
OP 68689 L 36 8   0 0 ----------000---11101100---00010----LLLLLLLL--------------------
TS 192 1
TS 266 1
TS 349 0
TS 4695 0
CT 157 4 A 44 4    0 0 0000100-------------------------------------AAAA----------------
CT 157 8 A 40 8    0 0 0000101---------------------------------AAAAAAAA----------------
CT 157 12 A 36 12    0 0 0000110-----------------------------AAAAAAAAAAAA----------------
CT 157 13 A 35 13    0 0 0000111----------------------------AAAAAAAAAAAAA----------------
CT 188 4 A 44 4    0 0 0000100111----------------------------------AAAA----------------
CT 188 8 A 40 8    0 0 0000101111------------------------------AAAAAAAA----------------
CT 188 12 A 36 12    0 0 0000110111--------------------------AAAAAAAAAAAA----------------
CT 188 13 A 35 13    0 0 0000111111-------------------------AAAAAAAAAAAAA----------------
CT 732 0    0 0    0 0 0000001---------------------------------------------------------
CT 898 4 A 44 4    0 0 1011100-------------------------------------AAAA----------------
CT 898 8 A 40.8    0 0 1011101---------------------------------AAAAAAAA----------------
CT 898 12 A 36 12    0 0 1011110-----------------------------AAAAAAAAAAAA----------------
CT 898 13 A 35 13    0 0 1011111----------------------------AAAAAAAAAAAAA----------------
CT 991 13 A 35 13    0 0 1011111111-------------------------AAAAAAAAAAAAA----------------
CT 1046 4 A 44 4    0 0 1100100-------------------------------------AAAA----------------
CT 1046 8 A 40 8    0 0 1100101---------------------------------AAAAAAAA----------------
CT 1046 12 A 36 12    0 0 1100110-----------------------------AAAAAAAAAAAA----------------
CT 1046 13 A 35 13    0 0 1100111----------------------------AAAAAAAAAAAAA----------------
CT 1212 4 A 44 4    0 0 0011100-------------------------0001--------AAAA----------------
CT 1212 8 A 40 8    0 0 0011101-------------------------0001----AAAAAAAA----------------
CT 1212 12 A 36 12    0 0 0011110-------------------------0001AAAAAAAAAAAA----------------
CT 1212 13 A 35 13    0 0 0011111-------------------------000AAAAAAAAAAAAA----------------
CT 2237 0    0 0    0 0 1011001---------------------------------------------------------
CT 2551 0    0 0    0 0 0011001-------------------------0001----------------------------
CT 7204 4 A 44 4    0 0 0010100-----------------------------1001----AAAA----------------
CT 7204 8 A 40 8    0 0 0010101-----------------------------1001AAAAAAAA----------------
CT 9307 0    0 0    0 0 0010001-----------------------------1001------------------------
```

APPENDIX D - SAMPLE MICROPROGRAMS

The program that processes ABMPL microprogram specifications is called the Alternative Based Microcode Generator. This program was constructed using the recursive descend method of compilation.

The compiler uses a symbol table to verify that items are declared before they are referenced. The statements declared in the instruction set are hashed and entered in the symbol table along with an identifier and any literal information. At microcode generation time the hash code is used to access the MIF to retrieve the corresponding microword. Statements in the instruction set are not checked for syntax, if a statement exist in the MIF, then it must be syntactically correct.

The predicates declared are also hashed and entered into the symbol table. The MIF is accessed using the hash code to retrieve the corresponding testing priority.

Cluster identifiers are entered into the symbol table along with their starting address in the microcode. This is done for branching and backpatching purposes during microcode generation.

Microprocedure identifiers are entered into the symbol table along with their parameter list. This done to verify actual parameters during procedure calls. No extra code is used to handle parameters since the cost is prohibitive. Parameter list are provided for information purposes only.

The output of the ABMPL compiler for two microprograms follows.

```
CONCORDIA UNIVERSITY COMPREHENSIVE MICROPROGRAMMING SYSTEM:  ALTERNATIVE BASED MICROCODE GENERATOR
 HOST: CYBER 170 NOS 1.3  TARGET: SEL32/75   82/07/29.    09.56.52.

MPROGRAM SQUARE(4096);
(*
   THIS PROGRAM SQUARES THE CONTENT OF REGISTER 6
   AND IF NECESSARY USES DOUBLE PRECISSION RETURNING
   THE MOST AND LEAST SIGNIFICANT PARTS IN
   REGISTERS 6 AND 7 RESPECTIVELY.
                                                                *)
 STORE REGFILE,              (* GENERAL REGISTERS 6,7 *)
       ALUZ,                 (* TEST ALU RESULT ZERO  *)
       HIREG,                (* REGISTER BANK SELECTOR *)
       T,                    (* MULTIPLICAND          *)
       DI,                   (* MULTIPLIER            *)
       S;                    (* MOST SIGNIFICANT PART  *)

 MPROCEDURE MULTIPLY(S,DI,T);
(*
   THE PROCEDURES EMPLOYS BOOTH'S ALGORITHM
   TO MULTIPLY SIGNED INTEGERS.
   S       ADDER-SHIFTER
   DI      MULTIPLIER-SHIFTER
   T       MULTIPLICAND
                                                                        *)
  STORE NALUZ,               (* TEST ALU RESULT NOT ZERO *)
        LIT,
        NCTR;                (* COUNTER                  *)

  ISET
   1) COCYCLE
        LIT:= $L1,
        NULL:= X'000000'&LIT AND DI
      END   <NALUZ>;
   2) COCYCLE
        LIT:= $L31,
        NCTR:= SELECT(0..7,LIT&X'000000')
      END;
   3) NCTR:= NCTR - 1;
   4) S:= 0;
   5) S:= S + T;
   6) S:= S - T;
   7) SHIFTC(RIGHT,1,S(0),S&DI);     (*DOUBLE ARITHM SHIFT*)

  PREDICATES
   1) NALUZ;
   2) NCTR = 0;

  PROCEDURE

   CLUSTER 1
    1) DO 2,4;1 GO 2.

   CLUSTER 2
```

D-3

```
CONCORDIA UNIVERSITY COMPREHENSIVE MICROPROGRAMMING SYSTEM:  ALTERNATIVE BASED MICROCODE GENERATOR  VI
 HOST: CYBER 170 NOS 1.3  TARGET: SEL32/75   82/07/29.   09.56.52.

    1) ON 1 DO 6;7;1 GO 3.            (* B'1'          *)
    2) ON -1 DO 7;1 GO 4.             (* B'0'          *)

   CLUSTER 3
    1) ON -2,1 DO 3,7;1 GO 3.         (* B'11'         *)
    2) ON -2,-1 DO 5;3,7;1 GO 4.      (* B'01'         *)
    3) ON 2 RETURN.

   CLUSTER 4
    1) ON -2,1 DO 6;3,7;1 GO 3.       (* B'10'         *)
    2) ON -2,-1 DO 3,7;1 GO 4.        (* B'00'         *)
    3) ON 2 RETURN.

 END;                      (* MULTIPLY *)

 ISET
  1) DI:= REGFILE[HIREG][$R6];
  2) T:= DI;
  3) MULTIPLY(S,DI,T);
  4) REGFILE[HIREG][$R6]:= S  <ALUZ>;
  5) REGFILE[HIREG][$R6]:= DI;
  6) REGFILE[HIREG][$R7]:= DI;

 PREDICATES
  1) ALUZ;

 PROGRAM

  CLUSTER 1
   1) DO 1;2;3;4 GO 2.

  CLUSTER 2
   1) ON 1 DO 5 RETURN.          (* RESULT <= (2**32)-1 *)
   2) ON -1 DO 6 RETURN.          (* RESULT >= 2**32 *)

END.
```

```
MICROCODE GENERATED FOLLOWS:

ADDR           BINARY REPRESENTATION                                               HEXADECIMAL REPRESENT
4096(X'1000')  0000000---000---11101100---00010----00011110111-----------------   0000EC0201F70000
4097(X'1001')  ----------01101111010000---00010----00000001--------------------   001BD00200100000
4098(X'1002')  0000100-------------------------------------0011----------------   0800000000030000
4099(X'1003')  1100100-------------------------------------0111----------------   C800000000070000
4100(X'1004')  0000-----------------------000010011----1001--------------------   0000000130900000
4101(X'1005')  ----------01101111010000---00010----00000001--------------------   001BD00200100000
4102(X'1006')  0000101---------------------------------00010101----------------   0A00000000150000
4103(X'1007')  ----------00000001010001----------------------------------------   0000510000000000
4104(X'1008')  0000-----------------------000010011----1001--------------------   0000000130900000
4105(X'1009')  ----------01101111010000---00010----00000001--------------------   001BD00200100000
4106(X'100A')  0000100-------------------------------------1011----------------   08000000000B0000
4107(X'100B')  1100101---------------------------------00010001----------------   CA00000000110000
4108(X'100C')  0010001-----------------------------1001------------------------   2200000009000000
4109(X'100D')  ----------00000000110001----------------------------------------   0000310000000000
4110(X'100E')  0000---011-----------------000010011----1001--------------------   00C0000130900000
4111(X'100F')  ----------01101111010000---00010----00000001--------------------   001BD00200100000
4112(X'1010')  0000100-------------------------------------0101----------------   0800000000050000
4113(X'1011')  0010001-----------------------------1001------------------------   2200000009000000
4114(X'1012')  0000---011-----------------000010011----1001--------------------   00C0000130900000
4115(X'1013')  ----------01101111010000---00010----00000001--------------------   001BD00200100000
4116(X'1014')  0000101---------------------------------00001011----------------   0A000000000B0000
4117(X'1015')  1100100-------------------------------------1010----------------   C8000000000A0000
4118(X'1016')  0010001-----------------------------1001------------------------   2200000009000000
4119(X'1017')  0000---011-----------------000010011----1001--------------------   00C0000130900000
4120(X'1018')  ----------01101111010000---00010----00000001--------------------   001BD00200100000
4121(X'1019')  0000100-------------------------------------0101----------------   0800000000050000
4122(X'101A')  0010001-----------------------------1001------------------------   2200000009000000
4123(X'101B')  ----------00000001010001----------------------------------------   0000510000000000
4124(X'101C')  0000---011-----------------000010011----1001--------------------   00C0000130900000
4125(X'101D')  ----------01101111010000---00010----00000001--------------------   001BD00200100000
4126(X'101E')  0000101---------------------------------00001011----------------   0A000000000B0000
4127(X'101F')  1111---------10000000110000---------0110------------------------   F004060006000000
4128(X'1020')  -------------01100001111----------------------------------------   00030F0000000000
4129(X'1021')  0000101111------------------------------00000000----------------   0BC0000000000000
4130(X'1022')  ----------000---11101000------------0110------------------------   000QE80006000000
4131(X'1023')  0000100-------------------------------------0100----------------   0800000000040000
4132(X'1024')  1011100-------------------------------------0110----------------   B800000000060000
4133(X'1025')  0000001------01100001000------------0111------------------------   0203080007000000
4134(X'1026')  0000001------01100001000------------0110------------------------   0203080006000000

ENTRY POINT OF MICROPROGRAM IS 4127(X'101F')
```

D-5

```
CONCORDIA UNIVERSITY COMPREHENSIVE MICROPROGRAMMING SYSTEM:  ALTERNATIVE BASED MICROCODE GENERATOR  VI
 HOST: CYBER 170 NOS 1.3  TARGET: SEL32/75   82/07/29.   10.05.20.

MPROGRAM FACTORIAL(4135);
(*
  THIS PROGRAM COMPUTES THE FACTORIAL OF THE CONTENT
  OF GENERAL REGISTER 6 AND IF NECESSARY USES DOUBLE
  PRECISSION, RETURNING THE MOST AND LEAST SIGNIFICANT
  PARTS IN REGISTERS 6 AND 7 RESPECTIVELY.
                                                    *)
  STORE REGFILE,      (* GENERAL REG 6,7 *)
        LIT,          (* LITERAL GENERATOR *)
        ALUSIGN,      (* SAVE SIGN OF ALU *)
        ALUZ,         (* TEST FOR ZERO IN ALU *)
        HIREG,             (* FLIP-FLOP GROUP 1 HIREG *)
        T,            (* MULTIPLICAND *)
        DI,           (* MULTIPLIER *)
        S;            (* MOST SIGNIFICANT PART *)

  MPROCEDURE MULTIPLY(S,DI,T);
   EXT 4096;
  END;

  ISET
   1) COCYCLE
       DI:= REGFILE[HIREG][$R6],
       ALUSIGN:= REGFILE[HIREG][$R6](0)
      END <ALUZ>;
   2) T:= REGFILE[HIREG][$R6] - 1 <ALUZ>;
   3) COCYCLE
       LIT:=$L255,  (* X'FF'*)
       T:= X'FFFFFF'&LIT + T    (* T:= T -1 *)
      END <ALUZ>;
   4) MULTIPLY(S,DI,T);
   5) NULL:= S <ALUZ>;
   6) REGFILE[HIREG][$R6]:= DI;
   7) REGFILE[HIREG][$R6]:= S <ALUZ>;
   8) REGFILE[HIREG][$R7]:= DI;
   9) REGFILE[HIREG][$R6]:= REGFILE[HIREG][$R6] + 1;

  PREDICATES
   1) ALUZ;
   2) ALUSIGN;

  PROGRAM

   CLUSTER 1
    1) DO 1 GO 2.

   CLUSTER 2
    1) ON 1 DO 9 RETURN.       (* FACTORIAL OF 0 *)
    2) ON 2 RETURN.          (* FACTORIAL OF NEGATIVE *)
    3) ON -1,-2 DO 2 GO 3.     (* DECREMENTED MULTIPLICAND *)
```

```
CONCORDIA UNIVERSITY COMPREHENSIVE MICROPROGRAMMING SYSTEM:  ALTERNATIVE BASED MICROCODE GENERATOR  VI
 HOST: CYBER 170 NOS 1.3  TARGET: SEL32/75   82/07/29.   10.05.21.

   CLUSTER 3
    1) ON 1 RETURN.             (* FACTORIAL OF 1 *)
    2) ON -1 DO 4;3 GO 4.         (* MULTIPLY; DECREMENT MULTIPLICAND *)
   CLUSTER 4
    1) ON 1 DO 7 GO 6.            (* MULTIPLICAND = 0, START TO QUIT *)
    2) ON -1 DO 5 GO 5.           (* CHECK MOST SIGNIFICANT PART *)
   CLUSTER 5
    1) ON 1 DO 4;3 GO 4.          (* PRODUCT <=(2**32)-1, PROCEED MULT & DECR *)
    2) ON -1 DO 7;8 RETURN.       (* PRODUCT >=(2**32), STOP MULT, RETURN *)
   CLUSTER 6
    1) ON 1 DO 6 RETURN.          (* FACTORIAL <=(2**32)-1 *)
    2) ON -1 DO 7 RETURN.         (* FACTORIAL >=(2**32) *)
 END.
```

```
MICROCODE GENERATED FOLLOWS:

ADDR            BINARY REPRESENTATION                                             HEXADECIMAL REPRESEI
4135(X'1027')   1111---------1000000011000010010----0110------------------------  F004061206000000
4136(X'1028')   0000100-------------------------------------1001----------------  0800000000090000
4137(X'1029')   1011100-------------------------------------1101----------------  B8000000000D0000
4138(X'102A')   0011001------------------------0001-----------------------------  3200000020000000
4139(X'102B')   0000------100---01001111000---------0110------------------------  00204F0006000000
4140(X'102C')   0000100-------------------------------------1110----------------  08000000000E0000
4141(X'102D')   0000001---100---01101000000---------0110------------------------  0220680006000000
4142(X'102E')   1011001---------------------------------------------------------  B200000000000000
4143(X'102F')   0000101111------------------------------00000000----------------  0BC0000000000000
4144(X'1030')   ----------11100000111111------------11111111--------------------  00383F000FF00000
4145(X'1031')   0000100-------------------------------------0010----------------  0800000000020000
4146(X'1032')   1011100-------------------------------------0101----------------  B800000000050000
4147(X'1033')   ----------000---11100000----------------------------------------  0000E00000000000
4148(X'1034')   0000100-------------------------------------0111----------------  0800000000070000
4149(X'1035')   ----------000---11101000------------0110------------------------  0000E80006000000
4150(X'1036')   0000100-------------------------------------1101----------------  08000000000D0000
4151(X'1037')   1011100-------------------------------------1010----------------  B8000000000A0000
4152(X'1038')   ----------000---11101000------------0110------------------------  0000E80006000000
4153(X'1039')   0000001------01100001000------------0111------------------------  0203080007000000
4154(X'103A')   0000101111------------------------------00000000----------------  0BC0000000000000
4155(X'103B')   ----------11100000111111------------11111111--------------------  00383F000FF00000
4156(X'103C')   0000100-------------------------------------0010----------------  0800000000020000
4157(X'103D')   1011100-------------------------------------1111----------------  B80000000000F0000
4158(X'103E')   0000001---000---11101000------------0110------------------------  0200E80006000000
4159(X'103F')   0000001------01100001000------------0110------------------------  0203080006000000
ENTRY POINT OF MICROPROGRAM IS 4135(X'1027')
```

APPENDIX E - MICROPROGRAM LOADER AND DRIVER

The jobs to load and drive the microprograms are shown in the following pages. The job to load consists of an assembly step, a catalog step to make the loader privileged, and an execution step to write into the WCS. In the assembly the doublewords corresponding to the microinstructions of the microprogram are stored in main memory using DATAD assembler directives. From main memory they are written into the WCS using a loop which includes a WWCS (write WCS) instruction. General register two points to the address of a doubleword in main memory, and register four points to the address in the WCS.

```
$JOB FACTL 710000 JUAN LINARES
$OPTION 2 5 15
$ASSIGN3 SLO=LP7A
$ALLOCATE 10000
$EXECUTE ASSEMBLE
         PROGRAM FACTL
R2       EQU 2
R3       EQU 3
R4       EQU 4
         BOUND 1D
MPROG    DATAD X'F004061206000000'
         DATAD X'0800000000090000'
         DATAD X'B8000000000D0000'
         DATAD X'3200000020000000'
         DATAD X'00204F0006000000'
         DATAD X'08000000000E0000'
         DATAD X'0220680006000000'
         DATAD X'B200000000000000'
         DATAD X'0BC0000000000000'
         DATAD X'00383F000FF00000'
         DATAD X'0800000000020000'
         DATAD X'B800000000050000'
         DATAD X'0000E00000000000'
         DATAD X'0800000000070000'
         DATAD X'0000E80006000000'
         DATAD X'08000000000D0000'
         DATAD X'B8000000000A0000'
         DATAD X'0000E80006000000'
```

```
          DATAD X'0203080007000000'
          DATAD X'0BC0000000000000'
          DATAD X'00383F000FF00000'
          DATAD X'0800000000002000'
          DATAD X'B8000000000F0000'
          DATAD X'0200E80006000000'
          DATAD X'0203080006000000'
INITWCS   DATAW X'0027'      WCS LOADING ADDRESS
START     BOUND 1W
          LEA R2,MPROG               FIRST DWORD OF MPG IN MAIN
          LEA R3,8*25+MPROG         LAST DWORD OF MPG IN MAIN
          LW R4,INITWCS
LOOP      WWCS R2,R4                 WCS[R4]:= MAIN[R2]
          ADI R2,8                   POINT NEXT DWORD OF MPG
          ADI R4,1                   POINT NEXT WCS LOCATION
          CAR R2,R3                  LAST DWORD OF MPG?
          BNE LOOP                   IF R2 <> R3 GOTO LOOP
          CALM X'55'                 QUIT NORMALLY
          END START
$IFT ABORT SKIPGO
$ASSIGN3 SLO=LP7A
$EXECUTE CATALOG
CATALOG FACTL BP P
$OPTION DUMP
$ASSIGN3 SLO=LP7A
$EXECUTE FACTL
$DEFNAME SKIPGO
$EOJ
$$
```

The job to drive the microprogram consist of an assembly step, and a debug step. In the assembly a number is loaded into register six and the microprogram is invoked through a JWCS (jump WCS) instruction. The debug step is performed through the DEBUG utility offered by SEL. In this step the utility is directed to take a snapshot of the registers and the area of main memory where the driver program resides. Snapshots are taken before the JWCS instruction, and after it. For both of the sample programs shown in appendix D the results observed were correct.

```
$JOB       MPRUN   710000 JUAN LINARES
$OPTION    2.5 15
$ASSIGN3   SLO=LP7A
```

```
$EXECUTE  ASSEMBLE
          PROGRAM MPRUN
NUMBER    DATAW   6               NUMBER TO BE PROCESSED
START     LW      6,NUMBER        LOAD REG6 WITH NUMBER
          JWCS    X'27'           JUMP TO ENTRY POINT IN WCS
QUIT      CALM    X'55'           QUIT NORMALLY
          END     START
$IFT      ABORT   SKIPGO
$ASSIGN3  SLO=LP7A
$EXECUTE  DEBUG
LOAD
SNAPSHOT  8,0,F
SNAPSHOT  C,0,F
START
$DEFNAME  SKIPGO
$EOJ
$$
```

The two jobs display in this appendix correspond to the loading and execution of the sample microprogram FACT, which computes the factorial of the contents of general register 6.