NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

A MICROPROGRAMMED INTERPRETER FOR CONCURRENT EUCLID

Kuarlall Lall

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

December 1988

ISBN 0-315-49095-0

ABSTRACT

A Microprogrammed Interpreter for
Concurrent Euclid

Kuarlall Lall

There are several methods of executing programs written in a high level language (HLL). The most widely used is to compile the programs into machine language. Another is to translate the programs into some intermediate form and then to execute that form interpretively. A third method is to directly execute either the HLL or the intermediate form.

This study was aimed at investigating the feasibility of directly executing the intermediate representation of the sequential features of Concurrent Euclid (CE) on the SEL 32/75 computer. The CE intermediate code was translated into Ecode, and a microprogrammed interpreter for Ecode was designed and implemented on the SEL, and benchmarked against the compiler. For the CPU-bound prime number algorithm Sieve of Eratosthenes, the interpreter was measured to be about twice as slow as the compiler. Ecode was then modified, and a new translator and interpreter designed and implemented. The same benchmark then yielded comparable results for both the interpreter and compiler. We project that further changes in Ecode design and hardware support would result in substantial Ecode efficiency gains.

ACKNOWLEDGEMENTS

I would like to express my sincerest thanks to to my thesis supervisor Dr. J. W. Atwood, without whose constant encouragement I probably would have given up on writing this thesis a long time ago. He was always readily available for consultation, and I appreciate the time and effort he put into guiding me through the project and constructing and correcting this thesis.

My thanks also go to Dr. H. Boom and Dr. T. Radhakrishnan for permitting me to incorporate preliminary work on this thesis into their Compiler Construction and Computer Architecture course requirements respectively. I would also like to thank M. Duarte and D. Hargreaves, who provided me with invaluable assistance in implementing this project on the SEL. I gratefully acknowledge the financial support of an FCAC operating grant which made it possible for me complete this project.

I dedicate this thesis to my wife Shamwattee whose patience and moral support helped me through all five years of it, and to my two baby daughters Kristina and Carolyn who are too young to understand why I had to limit our playing time during the final stages of this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

1.0 Introduction

There are several methods of executing programs written in a high level language. The most widely used is to compile the programs into machine language. Another is to translate the programs into some intermediate form and then to execute that form interpretively. A third method is to directly execute either the high level language or the intermediate form [HASS76].

A compiler is a computer program which accepts as input a program written a high level language (HLL) and produces as its primary output machine language code that will instruct some computer to produce results equivalent to those defined by the original HLL. Although there are many different ways to write one, all compilers perform two basic processes: analysis of the HLL source text, and synthesis of machine language instructions, or object text.

The use of intermediate languages as a convenient means of developing portable HLLs is now fairly standard. With this approach the compiler for language A compiles the source code into intermediate language I, which is usually pseudo machine language. For each machine that the language is to

be implemented on, there is either a program that converts I into assembler language for that machine or, alternatively, an interpreter may be written which executes the pseudo machine codes directly. The interpreter is usually considerably less efficient than a compiler because it carries the burden of intermediate code analysis in addition to execution.

The justification for the intermediate code approach is that portability of compilers is enhanced. The major sections of the compiler can be written in a high level, portable language to generate machine independent intermediate code, in which the complex HLL constructs have been translated into relatively simple and fewer intermediate code constructs. Portability is then obtained by writing what is a relatively small interpreter or compiler for the intermediate language to machine language.

Portability would be improved if the intermediate language could be executed directly as this would avoid either the step of converting the intermediate language, or the reduced performance resulting from interpreting it. With the availability of writable control store as an option on most minicomputers, there is merit in investigating the feasibility of direct execution of intermediate languages on a mini-computer. If feasible, then this represents an

efficient way to obtain language portability [COOP80].

1.1 Microprogrammed Interpreters

The use of microprogrammed interpreters to enhance high level language execution speed has been around for a long time. Papers describing the potential gains were quite common during the early 1970's [BROA75], and the Burroughs B1700/1800 machines were designed to directly execute the intermediate languages that were developed for different HLL's [COOP80]. To date, however, the author has been able to locate only a few published articles describing actual implementations of microprogrammed interpreters, and these are summarized below.

Broca and Mervin found gains of 12-75% in their implementation of a microcoded interpreter for Fortran compared with the Fortran H and G compilers on an IBM 360 computer system [BROC73]. Cooper has implemented a microprogrammed interpreter for subsets of the intermediate code generated from BCPL and Pascal [COOP80]. He did not report extensive benchmarking results but found that Wirth's prime number program [JENS75] ran three times as fast as the compiled version.

A microprogrammed interpreter for the intermediate language of Modula was designed by Habib and Yang using bit-slice AMD 2900 architecture [HABI81]. Schaeffer and Pratt investigated the effect of microcoding selected parts of a software interpreter for the intermediate language of UCSD Pascal. They reported significant improvements though not as high as they expected [SCHA83].

Gee et al have implemented a high performance Prolog engine by directly executing its intermediate form as generated by a Prolog compiler, on the VAX 8600. Their initial results indicate that their system is the fastest implementation of Prolog on a commercially available general purpose processor [GEE86]. Okuno et al implemented a microprogrammed version of a Lisp interpreter on a TAO/ELIS system and their results indicate that the speed of interpreted code of TAO is comparable to that of compiled codes of commercial Lisp machines [OKUN87].

1.2 Microprogrammed Interpreter For Concurrent Euclid

This thesis was aimed at investigating the feasibility of directly executing the intermediate representation of the sequential features of Concurrent Euclid (CE) on the SEL 32/75 computer. A CE compiler and source code for the VAX 11/780 and a user microprogrammable minicopmuter, the SEL

32/75, were available for this project. A SEL code generator was written and the Euclid compiler ported to the SEL. A translator was then designed and implemented to convert the intermediate code generated by the compiler into Ecode, a form more suitable for interpretation on the SEL. A microprogrammed interpreter for Ecode was designed and implemented on the SEL and benchmarked against the compiler. For the CPU-bound algorithm Sieve of Eratosthenes, the interpreter was measured to be about twice as slow as the compiler. Ecode was then modified, and a new translator and interpreter designed and implemented. The same benchmark yielded comparable results for both the interpreter and compiler.

1.3 Organization Of This Thesis

The remainder of this thesis describes this project. Chapter two introduces microprogramming and its applications while chapter three introduces compilers, interpreters, Concurrent Euclid and its intermediate code which is translated into Ecode for interpretation. Chapter four gives an overview of the SEL computer, its architecture and microengine. Chapter five describes the design of Ecode, referred to as Ecode-I, and chapter six the implementation of the original microprogrammed interpreter and its benchmarking. Chapter seven describes the modified Ecode,

referred to as Ecode-II, and the microcoded interpreter which yielded comparable results to the compiler. Chapter eight identifies topics for future study and concludes the project.

CHAPTER 2: MICROPROGRAMMING

2.0 Introduction

Digital computing systems have traditionally been described as being composed of five basic units: input, output, memory, arithmetic and logic, and control as shown in figure 2.1 below [REIG72].

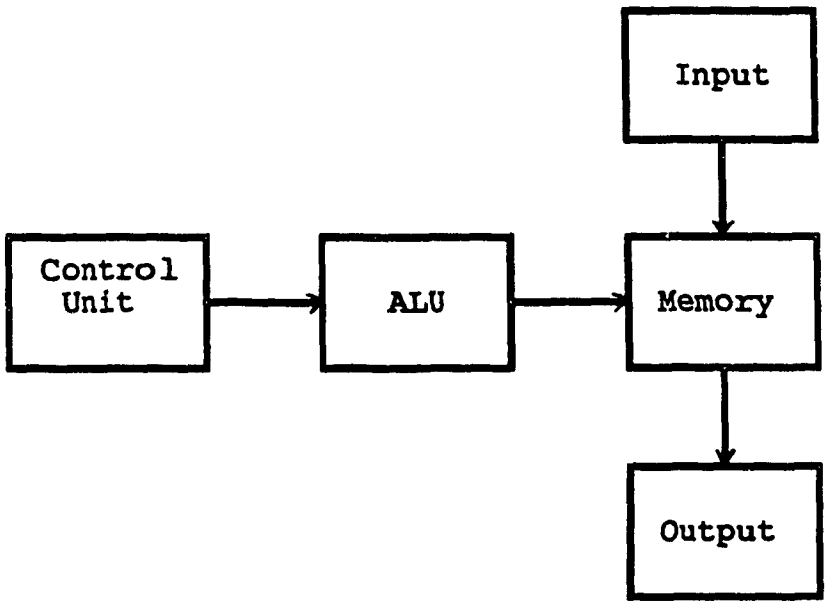


Figure 2.1 Five Basic Functional Units of a Digital Computing System (from RIEG72)

Machine instructions and data communication among the units (as indicated by the solid lines in figure 2.1) are generally well known and understood. The control signals (as indicated by the dashed lines) are generally less well

known and understood except by the system designer. These control signals generated in the control unit determine the information flow and timing of the system [REIG72].

Microprogramming was first proposed by professor M. V. Wilkes in 1951 as a systematic alternative to the rather ad-hoc method of designing the control system of a digital computer in use at that time [WILK69]. His thesis was that one can envision the control system of a computer as effecting a number of register-to-register transfers of information, some in sequence and some in parallel, in order to carry out the execution of a single machine instruction. The steps used to execute the instructions in a user machine can be thought of as constituting a program, called a microprogram [ROSI69]. Besides being a more structured approach to control system design, microprogramming introduced a large degree of flexibility in the design, implementation and maintenance of the instruction set of a computer.

2.1 Architectural Aspects Of Microprogramming

Juan Linares describes the hardware aspects of microprogramming quite nicely in his master's thesis [LINE82] and the following sections on control storage organization, microinstruction format and sequencing are

reproduced from his thesis.

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.

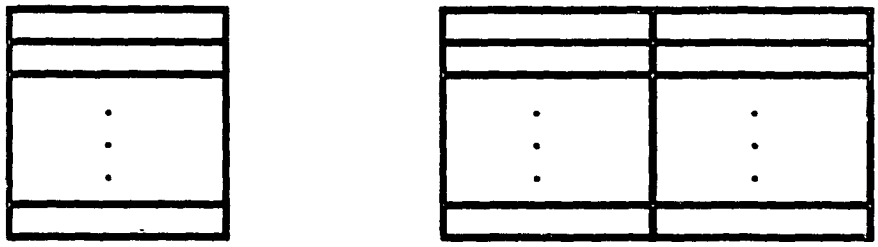
2.1.1 Control Storage Organization

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 B1700 in which microprograms are executed from an area of main memory [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.

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. 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. The memory array organization is illustrated in figure 2.2. Other organizations include the blocked, split structure and two level organization [LINE82].



One Microinstruction per Word

Two Microinstructions per Word

Figure 2.2: Memory Array Control Store Organization

2.1.2 Microinstruction Format

A microinstruction is merely a string of bits whose meaning 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

[RAUC80] although these designs refer to the extremes of a broad spectrum.

Vertical microinstructions effect single operations such as LOAD, STORE, and BRANCH; 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. These microinstructions have the potential advantage of efficient hardware utilization, but the optimization process is a difficult task. Figure 2.3 below illustrates vertical and horizontal microinstructions.

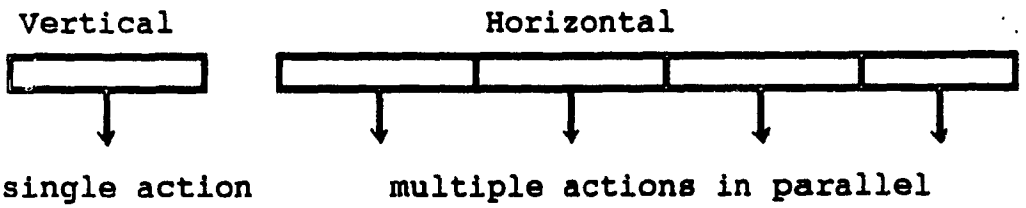


Figure 2.3: Vertical and Horizontal Microinstructions

2.1.3 Microinstruction Sequencing

The microinstructions sequencing mechanism is a great source of variability among microprogrammable machines, since they all have different and often inconvenient addressing

mechanisms [PERS77]. 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 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. In a parallel implementation, the fetch of the next microinstruction begins while the current one is being executed. The advantage of the serial approach is simplicity of realization, while the advantage of the parallel approach is the corresponding saving of time. Figure 2.4 illustrates the serial-parallel microinstruction fetch.

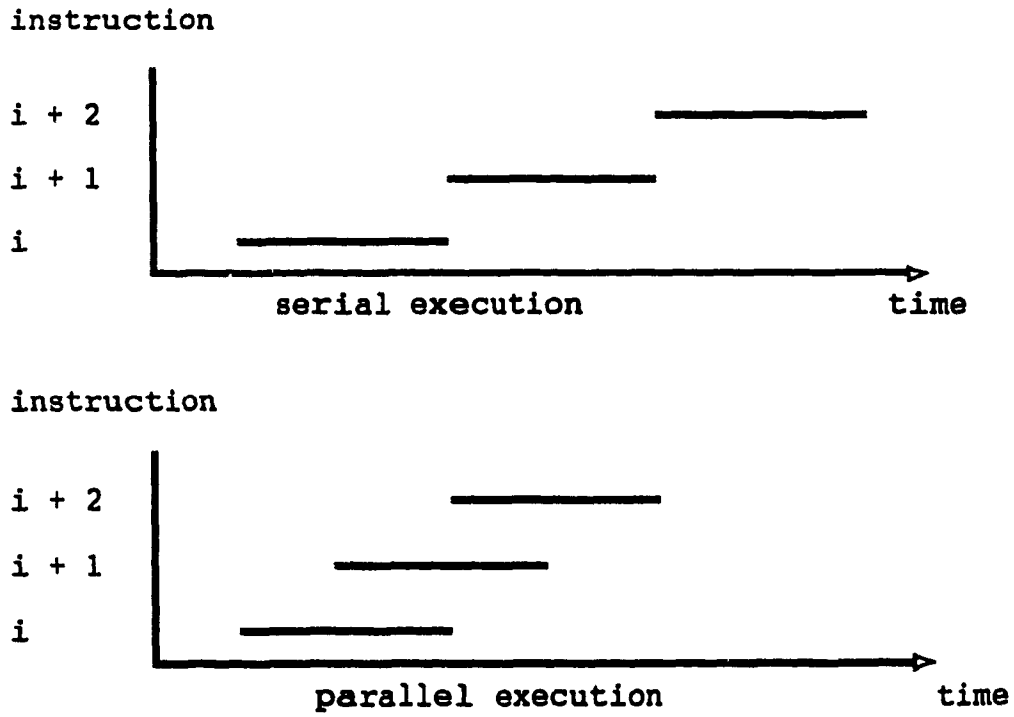


Figure 2.4: The Serial-Parallel Characteristics

The second aspect of sequencing is described by the number of minor clock cycles used to execute a microinstruction. 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. The advantage of monophase operation is the simplicity of realization, whereas the advantage of polyphase operation is that it allows better utilization of the resources at the expense of more complicated hardware.

2.2 Advantages Of Microprogramming Over Machine Language

The main advantages of microprograms over machine language programs are speed and accessability to the hardware data structures of the computer. Microprograms generally execute much faster than machine language programs because of several factors, namely:

- the ratio of control store speed over main memory
- the greater power of micro instructions over machine language instructions. A machine language instruction consist of more that one microinstruction.
- ability of microinstructions to allow parallel operations within the execution timing of a single instruction.
- direct interface of the microinstruction to the hardware, thereby eliminating the need to do memory fetches and decodes of all instructions.

These advantages of microprogramming have been utilized in a number of applications, some of which are described in the following sections.

2.3 Evolution of Microprogramming

Following Wilkes's initial investigation microprogramming received some attention during the 1950's, but it was not

until the 1960's that microprogramming was to be used significantly on a commercial scale. The main reason for this was that until the 1960's the simplicity and flexibility offered by microprogramming was more than offset by the tremendous overhead of a memory access for each microinstruction [RAUC80].

With the development of fast, inexpensive semiconductor memories there emerged an 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 the IBM 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 therefore does not imply the implementation of an entire instruction set of a computer in a microprogram. A machine instruction may be microprogrammed if its software implementation is too difficult, too inefficient, or if it is used so often as to be worth the effort of microprogrammming it.

The latest phase of of microprogramming is characterized by the appearance of user microprogrammable machines which provide tools to carry out research on the various aspects of microprograming. Advances in integrated circuits technology have led to the appearance of powerful microprocessors which have given great impulse to microprogramming. With the development of user microprogrammable machines, minicomputers and bit-sliced microprocessors the application of microprogramming spread to operating system support, fault diagnosis and special purpose systems, and support for high level language execution [RAUS80].

2.4 Applications of Microprogramming

As mentioned above, in addition to being an alternate way to design the control system of a computer and machine emulation, microprogramming can be used for operating systems support, fault diagnosis and special purpose systems, and to execute high level languages.

2.4.1 Operating systems support

Microprogramming can be used to assist the implementation of operating systems in two basic ways. First, by direct implementation of primitives in microcode and second,

microprograms can support primitives by implementing a suitable virtual machine on which the primitives can be executed. Primitives which are ideal candidates for miprogrammed implementation include bit manipulation operations, search routines, process synchronization, interprocess communication and protection, and interrupt handling. On the Burroughs B1700 computer a microprogrammed kernel handles time critical operating systems functions such as interrupt handling, scheduling, I/O processing and virtual memory management [WILK72].

2.4.2 Fault Diagnosis And Special-Purpose Systems

Microdiagnostics are microprograms that diagnose system hardware to detect and locate hardware faults. In many computer systems, especially real time systems, it is necessary to continue operation even in the presence of hardware failures. Microdiagnostics can generally access all CPU resources and locate hardware faults with a higher resolution and much faster than other methods, and do not require extensive use of main memory. Microprograms are ideally suited for special purpose, CPU intensive applications. These include signal processing, computer graphics, numerical algorithm implementation, and implementation of special systems for research and development purposes.

2.4.3 High Level Language Execution

The principal ways of using microprogramming to support high level language processing are: compile the high level language directly into microcode, microprogram critical sections of the high level language program, and use different target codes and microcoded interpreters for each language.

The first approach will generally give the most efficient implementation. In general this approach involves a much more complex operating system and several problems that occur at the machine code level show up again at the microcode level [BROA75]. In spite of these complications such languages and compilers have been designed and implemented on minicomputers and microcomputers. Fagin et al describe the compilation of Prolog directly into microcode, resulting in the fastest functioning Prolog system known to them [FAGI85]. This approach continues to be a main focus of microprogramming research in the academic community [SHRI81].

The second approach offers a way to improve the execution speed of a given high level language program by analysis of the program to determine the sections where most of the CPU execution time is used, and microcoding of these parts of

the program. Time critical applications such as real time processing and operating systems are good candidates for this type of support.

The third approach generally involves translating the high level language into an intermediate form and interpreting this intermediate code with a microcoded interpreter. The theory is that the greater speed of the microprogram will offset the higher overheads associated with interpreters. The advantage of this scheme is that it is not as complicated as the first and it is more general that the second. The microprogrammed interpreter for UCSD Pascal on the PDP-11 is an example of this [SCHA83].

2.5 Current Aspects Of Microprogramming

The current microprogramming interest in the academic community can be grouped into three general categories, namely, the automatic generation of correct, compact microcode from a high level language for different target machines, computer architecture design, and development of microprogramming tools.

By far the greatest emphasis is on the first category, with considerable material reported in the literature on projects on high level microprogramming languages (HLMLs), microcode

compilers, microprogram generation systems for retargetable implementations, hardware description languages (HDLs), and microcode compaction and verification schemes [SIGM86], [SIGM85]. The general objective is a microcode generation system which accepts as input an HLML and an HDL for a given machine, and outputs a correct and optimized microprogram for that given target machine.

Microprogramming research in computer architecture design is geared towards the development of processors for a variety of general and special purpose applications. Control Data Corporation is applying microprogramming techniques to the development of a multiple instruction set architecture processor using VLSI and CMOS technology [WILK84]. Patt et al report working on microarchitectures for implementing high performance computing engines [PATT85]. DuBose et al describe the initial design of a microcoded RISC-type machine, MIRIS, under development at George Mason University. The basic difference between MIRIS and other research prototype RISC machines is that the control of MIRIS is microcoded while the others are hardwired [DuBO86].

The increased use of microprogramming in recent years has created a need for sophisticated tools to support the development of microprograms. The literature reports on the development of interactive high level debuggers for

microprograms, and microprogram simulators for given architectures. The latest development is the automatic tool generation process which accepts as input a description of the microarchitecture in an HDL and generates as output an assembler, linkage editor and simulator tailored for that architecture [TRAC85].

CHAPTER 3: COMPILERS, INTERPRETERS AND CONCURRENT EUCLID

3.0 Compilers And Interpreters

A compiler is a translator which converts an input "source" language into an output "object" language which is recognizable to a specific computer hardware configuration. The translation typically occurs in three phases. The first includes syntactic analysis of the source program to guarantee its correctness and tabulation of all symbols; the second consists of a semantic analysis which converts the source statements into an intermediate text form. The third phase, referred to as code generation, converts the intermediate text into machine code for a particular hardware system.

An interpreter differs from a compiler in that it does not generate machine code but executes the intermediate text. It is usually considerably less efficient than a compiler because it carries the burden of intermediate code analysis in addition to execution. P. J. Brown [BROW81] explains this difference quite nicely with the following analogy. Assume you are an English speaker who does not understand French very well, and you are given some instructions in French to do a certain job. Assume further that you are a bit stupid, like a computer, and do not remember anything

unless you write it down, and then later read back what you have written.

The simplest way of executing the French instructions is to take each one in sequence, figure out what it means, and then obey the instructions. Thus, performing an instruction consists of two stages: decoding and action. The disadvantage of this is that if an instruction is repeated several times you have to repeat the decoding of the French instructions equally many times -- do not forget you are too stupid to remember them automatically. This suggests an alternative approach: first decode all the French instructions into English and write them down; then follow the English instructions.

The second approach is initially more time consuming, because translating into proper written English is more of an effort that figuring out the French instructions in your head. However, it becomes faster overall if the instructions are to be repeated. An interpreter corresponds to the first approach and a compiler to the second approach.

Because of the huge overheads of interpretation of a source language almost all compilers and most interpreters translate the source language into an intermediate form

which is easier to decode. This intermediate code could take a variety of forms. At one extreme it could be machine code, as would be for a compiler; at the other it could be almost the same as the source language, as would be the case for an (almost) pure interpreter. As the intermediate form moves away from the source language towards machine language the compiler gets steadily bigger but the users' programs run steadily faster. There is a spectrum of possiblilities between these two extremes, and real production compilers and interpreters lie all along the spectrum. Figure 3.1 below shows the steps taken by a compiler and interpreter.

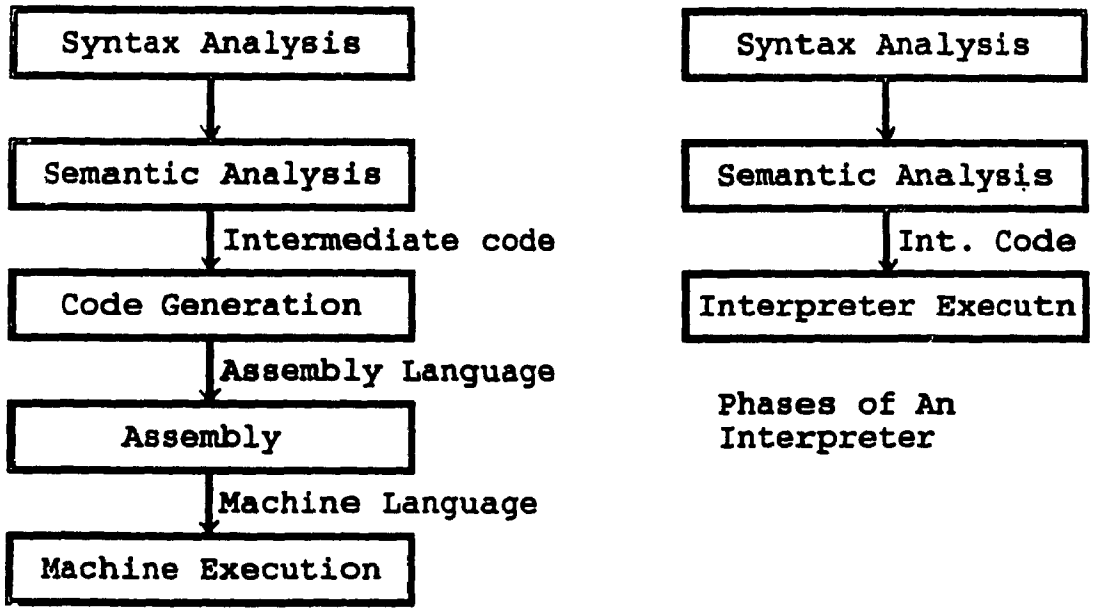


Figure 3.1: The Phases of a Compiler and Interpreter

3.1 Concurrent Euclid

Concurrent Euclid (CE) was designed to support implementation of highly reliable, high performance software such as compilers and operating systems. CE is based on Pascal and borrows Pascal's elegant data structures. Various features of Pascal were "purified" to allow easier verification; for example, in CE functions are prevented by the compiler from having side effects. The major features CE adds to Pascal are:

1. Separate compilation - procedures, functions and modules can be separately compiled and later linked together.
2. Modules - a module is the syntactic packing of data structures with the procedures and functions that access the data.
3. Concurrency - Monitors and processes are supported. There is a SIGNAL and WAIT statement. A BUSY statement allows CE to be used as a simulation language.
4. Control of Scope - names of variables, types, etc., are not automatically inherited by scope. Import and export lists are used to control the scope of names.
5. Systems programming constructs - These include

variables at absolute addresses. Such variables can be device registers in computers with memory mapped I/O.

There are some Pascal features, such as enumerated types, that CE does not support. CE does not allow procedures and functions to be nested inside procedures and functions. More details on CE are presented in Appendix I and a complete description given in [HOLT83].

3.2 Concurrent Euclid Compiler

The CE compiler makes four passes over the source input and its intermediate forms. The first three of these (the parser, semantic analyzer and storage allocator) are machine independent. The fourth, the code generator, is the only one that has to be changed to port the compiler to another machine. The intermediate code which will be transformed and interpreted by the microprogrammed interpreter is the output of the storage allocation pass.

A complete formal description of CE is given in [HOLT83]. Below is an example of a CE implementation of a stack. Two operations Push and Pop are defined on a data structure called Table. Push adds an item to Table and Pop returns the item most recently added to Table. The initially block

sets the number of items in Table to zero at the start of execution.

```
var stack:
    module
    exports (push, pop)
    const depth := 1..10
    var top: 0..depth
    var table: array 1..depth of signedint
procedure push(i:signedint)=
    imports (var top, var table)
    begin
        top := top + 1
        table(top) := i
    end push
procedure pop(var i:signedint) =
    imports (var top, var table)
    begin
        i := table(top)
        top := top - 1
    end pop
initially
   imports (var top)
   begin
       top := 0
   end
end module
```

3.3 CE Intermediate Code

The intermediate form generated as output of the storage allocator pass of the compiler is a string of tokens. The intermediate representation of the procedure "push" is shown below with comments.

```
Intermediate Representation                          Comment

aRoutineIndex  0
aIdentText 4 push                                    procedure push
aNewline  9
aBegin                                               begin block
aNewline  10
aDataDescriptor  162    1  0  0 2   0 0   top
aAssign                                              =
aDataDescriptor  162    1  0  0 2   0 0   top
aDataDescriptor    1  127  1  0 1   0 1   1
aAdd                                                 +
aEndExpression
aNewLine  11
aDataDescriptor  162    1  3  0 20  0 2   table
aSubs                                                start subscript
aDataDescriptor  162    1  0  0 2   0 0   top
aEndExpression
aEndSubs                                             end subscript
```

```
aDataDescriptor    1  127  1  0 1   0 1   lower bound
aDataDescriptor    1  127  1  0 1   0 1   upper bound-1
aDataDescriptor    1  127  1  0 1   0 9   size of item
aDataDescriptor  129  127  0  0 2   0 0   array attributes
aAssign                                   =
aDataDescriptor  162    2  0  0 2  -1 -4   I
aEndExpression
aNewLine  12
aEndBegin                                 end begin block
aNewLine  13
```

An aNewline token refers to the source line number that generated the code following it. The most complex structure in the intermediate language is the specification of data objects. A simple data object is represented by an "aDataDescriptor" token which has five fields as shown below:

Status	Base	Rep.	Value	Displ.

The status field is a bit encoded string indicating the addressing and alignment of the operand. The base field describes the location of the operand. The possibilities are: on runtime stack, in global read/write storage area, immediate operand in the descriptor itself, or in a register. The representation and value fields determine the

sign and size of the operand, that is, unsigned byte, signed long (or 4 bytes), array etc. The displacement field gives the displacement of the operand from one of the bases described in the base field. In the case of an immediate operand this field contains the actual value of the operand. The displacement and value fields are both 32 bits long and are implemented as two 16 bit values. The other three fields are 16 bits long.

As an example, consider the data descriptor for the variable "top" in the preceding example of the CE module "stack". The data descriptor is:

```
aDataDescriptor  162  1  0   0 2  0 0
```

The values and interpretations of the corresponding fields are as follows:

Status	= 162 -	bit 1 is set - indicates indirect addressing bit 5 is set - indicates operand has a lexic base bit 7 is set - operand is aligned on a two byte boundary
Base	= 1 -	indicates operand is on run-time stack

Representation	= 0	- representation an value fields
Value	= 0 2	- indicate operand is signed 16 bit integer
Displacement	= 0 0	- zero indicates that the operand starts at the base address.

An array element requires at least six data descriptors for its specification. These are:

- start location and size of array - 1 data descriptor
- subscript - at least 1 data descriptor
- lower bound - 1 data descriptor
- upper bound-1 - 1 data descriptor
- size of item - 1 data descriptor
- attributes of item (eg. signed/unsigned) - 1 data descriptor.

A more detailed description of this intermediate code is given in Appendix I.

CHAPTER 4: INTRODUCTION TO THE SEL

4.0 Introduction

The SEL 32/75 is a high speed, general purpose, digital computer system. It is designed for a variety of scientific, data acquisition and real time applications. A basic system includes a central processing unit, main memory subsystem, and microprogrammed I/O controllers.

The CPU has a large instruction set that includes fixed and floating point arithmetic instructions. A special lookahead feature enables the CPU to overlap instruction execution with memory accessing, thereby reducing program execution time. The main memory of 16 megabytes can consist of up to 16 modules of 64K bytes each on each of up to 16 memory busses. Memory can be shared by up to 20 CPU's and their associated I/O processors [SEL1].

The SEL 32 series computers use a microprogrammed control section (CROM) to decode and execute machine instructions. The writable control store (WCS) option consists of one or two 64 x 2K high speed random access memory boards which provide a physical extension of the control store. This feature allows the user to tailor the machine to accommodate any special user needs.

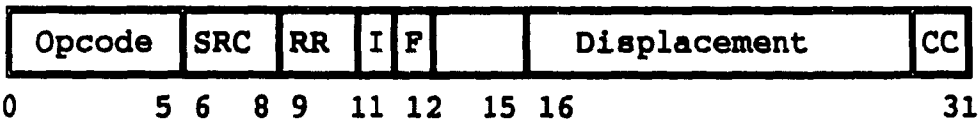
4.1 SEL Macroarchitecture

4.1.1 Registers

The SEL 32/75 has eight general purpose registers (GPR's) for use by the assembly language programmer for arithmetic, logical and shift operations. Three of the eight GPR's, R0, R1, R2, can also be used for indexing operations. Register R0 can also be used as a link register, and R4 can be used as a mask register.

4.1.2 Addressing modes

The general format of a memory reference instruction is shown below.



bits 0..5 - opcode

bits 6..8 - source register

bits 9..10 - index register

bit 11 - indirect addressing

bit 12 - F bit. memory addressing

bits 16..31 - displacement or literal

bits 30..31 - C bits. byte/halfword/word/doubleword addressing

Bits 9..31 have the same format in every memory reference instruction, regardless of whether the effective address is used for storage or retreival, as an indirect address, or to alter program flow. The format of the F and C bits have been selected so that any specified data type byte, 16 bit halfword, 32 bit word, or 64 bit doubleword can be conveniently referenced. The possible combinations of F and C bits are as shown in table 4.1 below:

F bit (12)	C bits (30,31)	Data type
0	0 0	32 bit word
0	0 1	16 bit half word. bits 0..15
0	1 0	64 bit doubleword
0	1 1	16 bit half word.bits 16..31
1	0 0	byte 0. bits 0..7
1	0 1	byte 1. bits 8..15
1	1 0	byte 2. bits 16..23
1	1 1	byte 3. bits 24..31

Table 4.1: SEL F and C Bits Addressing

The following addressing modes are provided:

1. Direct addressing - the effective memory address is taken directly from bits 13..31 of the memory reference instruction.
2. Indexed addressing - bits 13..31 are used to produce a memory address by adding it to the contents of the register specified by bits 9..10. Only registers

1,2 and 3 can be used as index registers.

3. Indirect addressing - the address of the operand is contained in the memory word specified by adding the contents of bits 13..31 to the contents of the register specified in bits 9..10.
4. Immediate addressing - the operand is in bits 16..31 of the instruction.
5. Register addressing - the operand is in a register specified by bits 6..8.

4.2 Instruction Repertoire

The functional classification and number of instructions for the SEL 32/75 computer are shown in table 4.2. A complete list of the SEL 32/75 instructions is given in [SEL1].

4.3 SEL Assembler Directives

The Ecode programs interpreted by the microprogrammed interpreter are generated in SEL assembly language which are translated by the SEL assembler into machine code. A partial description of the SEL assembly language directives used in Ecode generation is given below, and a complete list in Appendix II.

Directive/ Instruction		Comment
BOUND	N	forces the program counter to an N byte boundary; for example N = 4 indicates fullword boundary and N = 2 indicates halfword boundary.
GEN	N/B	define N bits of memory with value B; for example GEN 8/1,8/2,8/3,8/4 generates the bit configuration: 0000 0001 0000 0010 0000 0011 0000 0100
LABEL	EQU VALUE	equals tag; equates LABEL with VALUE

classification	number
Fixed point arithmetic	30
floating point arithmetic	8
boolean	17
load/store	29
bit manipulation	8
zero operand	5
shift	13
interrupt	13
compare	11
branch	9
register transfer	13
input/output	10
control	16
hardware memory management	4
writable control store	3
total	189

Table 4.2: SEL 32/75 instructions by category.

4.4 SEL Microengine

The operation of the SEL 32 computer is controlled by the central processing unit (CPU). In the CPU, the controlling hardware which executes the firmware (microprogram) is referred to as the microengine. Figure 4.1 presents a block diagram of the SEL microengine.

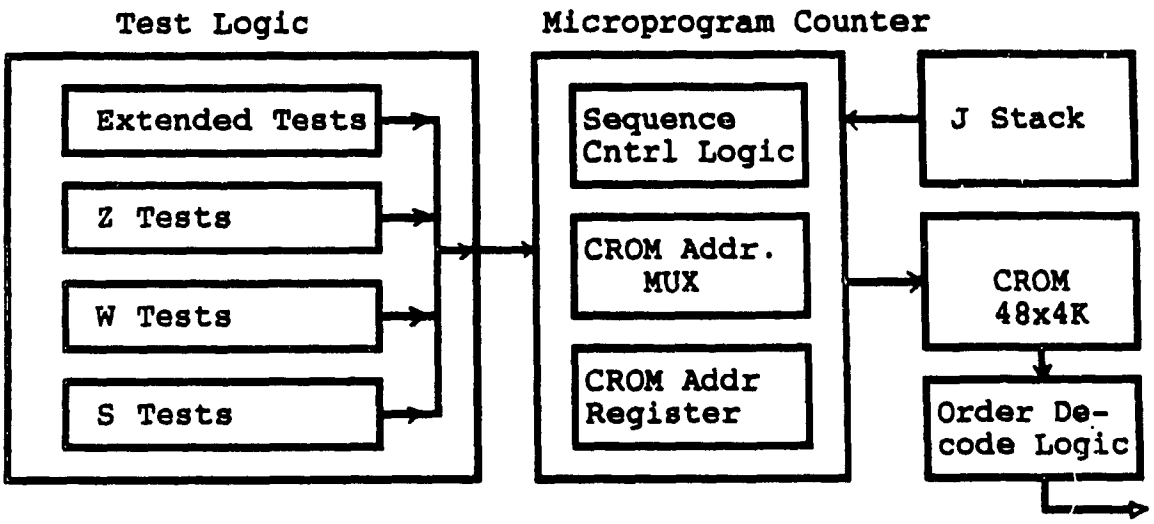


Figure 4.1: Block Diagram of SEL Microengine

The microengine consists of the following hardware sections:

1. Control Store (CROM) - consists of several read only memories used to store the microprograms.
2. Test Logic - the basic tests are the first part of the microinstruction to be executed. All basic tests must be completed before execution of the

microinstruction orders.

3. Microprogram counter - consists of several hardware sections which are used to select from a number of sources the next CROM address to be used by the microprogram.
4. Order decode stack - the last part of the microinstruction to be executed is the microinstruction orders (or operations). Orders are decoded and executed by this hardware.
5. J stack - a 4 x 13 bit register stack that acts as a last in, first out microprogram address stack. This can be used to implement microsubroutine calls.

4.5 Timing

Instruction execution within the microengine generally requires two cycles, the first being the CROM cycle and the second is the CREG cycle. Each cycle is 150 nanoseconds long thereby requiring a total of 300 nanoseconds to complete an instruction. During the first 150 nanoseconds the basic tests and sequencing are done; the second 150 nanoseconds execute all orders that the microinstruction directs. Although each microinstruction requires 300 nanoseconds to execute fully, one microinstruction can be completed every 150 nanoseconds by overlapping the CROM cycle of the second instruction with the CREG cycle of the

first instruction as shown in figure 4.2.

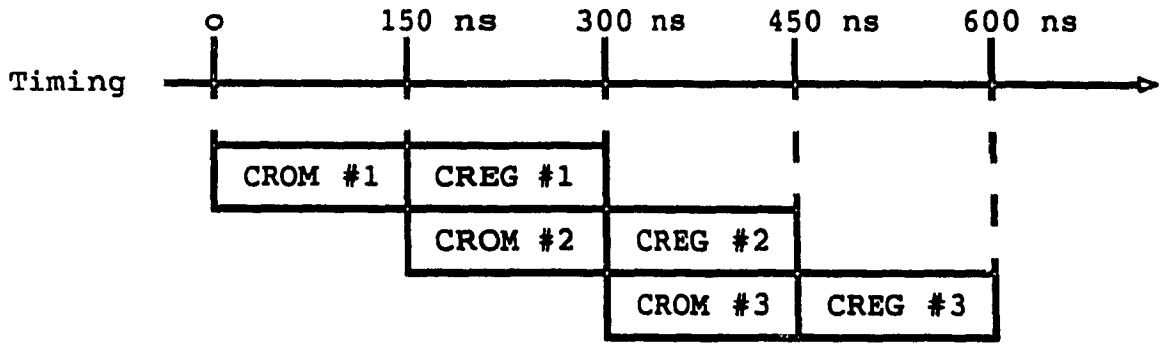


Figure 4.2: Microinstruction Timing

4.6 SEL Data Structure

The SEL data structure consists of 32 x 32 general file registers, hardware registers, and two multiplexors organized around an Arithmetic and Logic Unit and a 256 x 32 bit local store. The hardware registers are used for SELBUS communications, temporary storage, and shifting. Figure 4.3 shows a diagram of the SEL data structure. Unlike the machine language programmer who has access to only eight general purpose registers, the microprogrammer can directly access the entire SEL data structure. The data structure is presented in the following order:

1. Arithmetic and Logic unit - The ALU is a two-input 32-bit Arithmetic and Logical unit, utilizing four

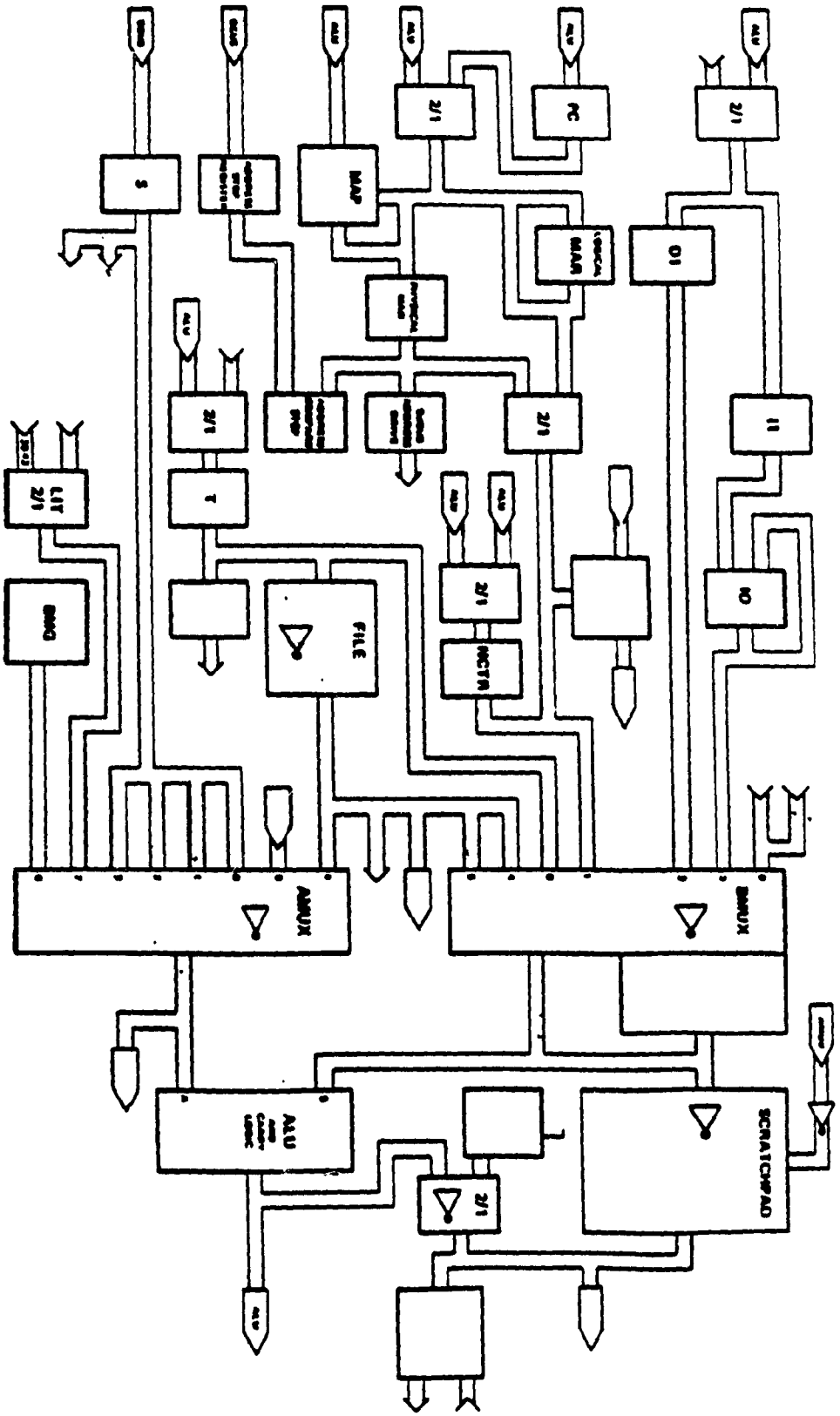


Figure 4.3: The SEL Data Structure

lookahead carry generators for increased speed of operation. The inputs to the ALU are selected by the A-Mux and B-Mux. The output destination of the ALU may or may not be specified. If not specified the output of the ALU is used for testing purposes only. The output of the ALU can be distributed to any of the registers and WCS output data.

2. A-Multiplexor (AMUX) - selects input into ALU
3. B-Multiplexor (BMUX) - selects input into ALU
4. Literal Generator - generates an 8 bit constant
5. General file registers (FILE) - 32 x 32 bit general purpose registers organized in two banks of 16 registers each.
6. Memory address register (MAR) - 24 bit register used to address main memory
7. Program counter register (PC) - a 22 bit counter used to address the next instruction to be executed.
8. N-Counter register (NCTR) - 8 bit binary counter. Can be incremented or decremented in the CREG cycle or decremented in the CROM cycle
9. Shift register (S) - 32 bit register used for shifting, either by nibble (4 bits) or by bits.
10. Temporary register (T) - 32 bit register used to

temporarily hold all data to be stored in the general purpose registers.

11. Data input register (DI) - 32 bit register used to receive operands from memory or data and status from I/O processors.
12. Instruction decode register (I0) - 32 bit register containing the current instruction being executed.
13. Instruction pipeline register (I1) - 32 bit register used to receive macro instructions as they return from memory. This register usually contains the next instruction to be executed.
14. Local store (SCRATCH) - 256 x 32 bit RAM storage for fast access data storage.
15. Bit mask generator(BMG) - generates 32 bit masks for bit manipulation instructions.

4.7 CPU Microword

The full CPU microword is 64 bits of which only 48 are directly associated with CPU operations, the remaining 16 bits are used for the optional high speed floating point unit. The microword is divided into 13 fields, each of which define tests and operations to be executed in parallel. These fields are described below.

1. Primary test field (T-field) - specifies 16 basic tests which are decoded during the CROM timing cycle
2. Sequence field (S-field) - specifies the address sequencing
3. Control field (M-field) - executed in CROM cycle
4. A-Mux field (A-field) - selects source of A input to the ALU
5. B-Mux field (B-field) - selects source of B input to the ALU
6. ALU field (+ field) - selects the ALU function to be performed
7. Destination field (D-field) - selects destination for ALU output
8. File register field(R-field) - selects one of 16 registers
9. Y-Order field (Y-field) - the five order fields
10. X-order field (X-field) - X, Y, P, C, and H
11. P-Order field (P-field) - define specific
12. C-Order field (C-field) - instructions to be carried out on the input/output

13.H-Order field (H-field) - of the ALU.

A complete description of the SEL writable control store can be obtained from [SEL2].

CHAPTER 5: DESIGN OF ECODE-I

5.0 Introduction

An interpreter generally executes the intermediate code directly instead of translating it into machine language as is done in a compiler. To interpret Concurrent Euclid a translator was written to convert the intermediate code generated by the CE compiler into Ecode-I, a form more suited for interpretation on the SEL. The design of Ecode-I incorporated inputs from the CE intermediate code and the SEL microarchitecture, as shown in figure 5.1 below. The format of Ecode-I is described in the following sections.

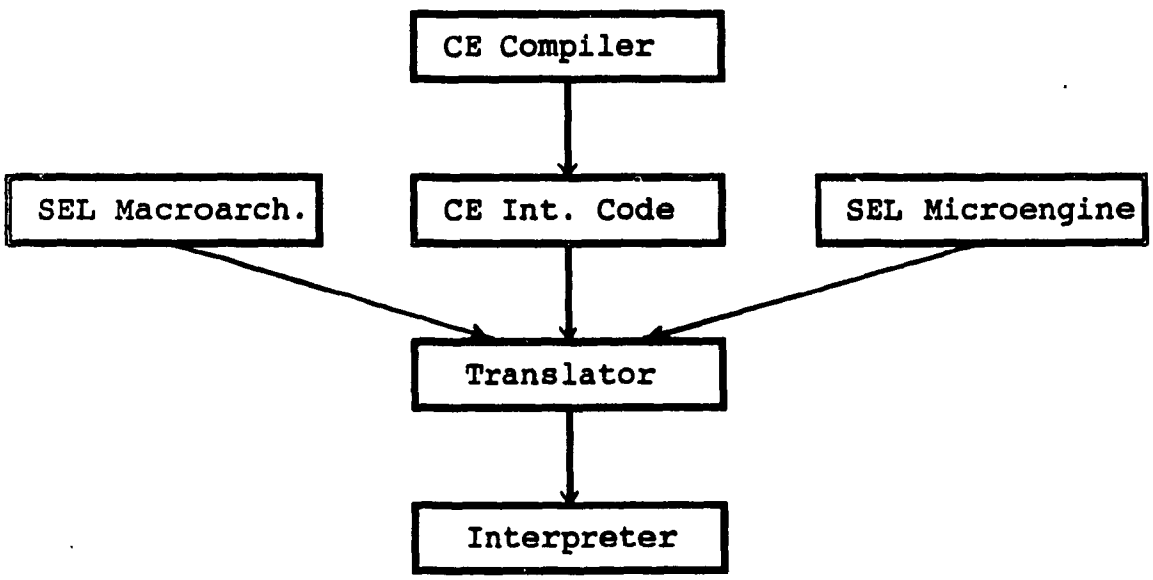


Figure 5.1: Design of Ecode-I

5.1 Instruction Set

The functional classification and number of instructions in Ecode-I is shown in table 5.1 below.

Instruction Category	Number
Branch	9
Fixed Point Arithmetic	12
Logical	4
Set Manipulation	2
Shift	2
Short Arithmetic	120
Short Logical	60
Short Set Manipulation	30
Miscellaneous	14
Total	253

Table 5.1: Ecode-I Instructions by Category

Each instruction consists of an opcode followed by one or more operands. The opcode and the first operand occupy a word of memory. Subsequent operands, if any, occupy additional words. The opcodes indicate the number of operands in the instruction, while the location of each operand is determined at run-time by the interpreter. The length of the instruction depends on the number of operands. The destination operand is always specified after the source operand as this allows fetching of the operands and execution of the operation to be performed in parallel with decoding of the destination address. For example, the

equation "A = B + C" is translated into the Ecode-I instruction "Add3, Operand B, Operand C, Operand A". During execution, operands B and C are fetched and the addition performed while the address of operand A is computed. Sample instructions are illustrated in table 5.2 and a complete list is given in Appendix III.

Instruction	# of Operands	Comment
Branch Equal	1	Branch to specified address if the previous comparison was equal.
Shift Left	2	Shifts operand left the number of bits specified by the second operand
Add2	2	Adds two operands and stores the result in the second operand.
Subtract3	3	Subtracts one operand from the other and stores the result in a third operand.
Logical And3	3	Performs a logical AND of two operands and stores the result in a third operand.
Set Difference2	2	Performs a set subtraction on the two operands and stores the result in the second operand.

Table 5.2: Sample Ecode-I Instructions

5.2 Optimizations

The following optimizations were implemented in the translator to improve the efficiency of Ecode-I:

1. A = 0 generates "Zero A"
2. A = A + A generates "Shift Left Arithmetic A"
3. A = A / 2 generates "Shift Right Arithmetic A"
4. A = -A generates "Negate A"
5. A = -B generates "Minus Assign A B"
6. A = B where both A and B are non-scalar, that is a table or an array, generates "Non-Scalar Assign A B " instead of a loop.
7. A = A + i where i is a literal integer between 0 and 15, generates "Short Add2i A"

The "Short" instruction was taken from N. Wirth's implementation of Lilith: A modula Machine [WIRT84], in which literal values between 1 and 15 in were embedded in the opcode itself, thereby shortening the instruction. For example A = A + 2 generates "Short Add22 A" where the value of the literal two is the four least significant bits of the eight bit opcode. There are 220 "Short" instructions in Ecode-I, used for arithmetic, logical and set manipulation instructions.

5.3 Memory Referencing

The operand specification is complicated by the SEL addressing and memory logic, and the SEL 32 bit word length. The SEL supports 64 bit doubleword, 32 bit word, 16 bit halfword, and 8 bit byte addressing according to the F and C bits (bits 30 and 31) in the address as shown in table 4.1 which is duplicated below:

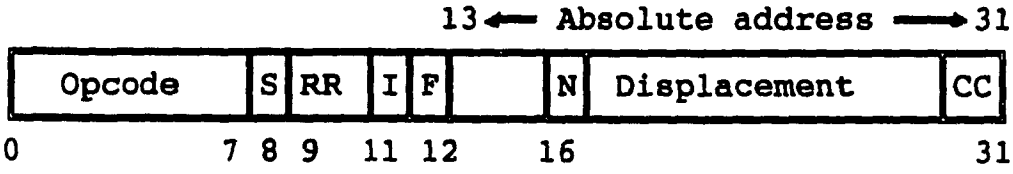
F bit (12)	C bits (30,31)	Data type
0	0 0	32 bit word
0	0 1	16 bit half word. bits 0..15
0	1 0	64 bit doubleword
0	1 1	16 bit half word.bits 16..31
1	0 0	byte 0. bits 0..7
1	0 1	byte 1. bits 8..15
1	1 0	byte 2. bits 16..23
1	1 1	byte 3. bits 24..31

As indicated by the C bits above, the address of a 16 bit quantity is always one byte greater than the actual address. The SEL memory reference logic recognizes this convention and reads the appropriate halfword. The addresses generated by the storage allocation pass of the CE compiler are machine independent and do not compensate for this addressing scheme. In Ecode-I generation, all known halfword addresses are incremented by one byte, and the F bit forced to a one or zero depending on the size of the operand. If the address is unknown at Ecode-I generation time, then it will be generated at run time, in which case

the SEL memory reference logic automatically stores the adjusted value.

Memory reads on the SEL do not sign extend 16 bit half words or 8 bit byte operands, they are zero filled by the memory reference logic. In CE there are no 64 bit operands, 32 bit operands are always signed, 16 bit operands could be either signed or unsigned and 8 bit operands are always unsigned. The memory read returns 32 bit and 8 bit operands in the correct format from memory, with the most significant bits zero filled for 8 bit operands. The sign bit (bit 8) in Ecode-I is used to indicate whether 16 bit operands should be sign extended or not.

5.3 Operand Specification



Opcode - bits 0..7 Indicate the instruction opcode

S - bit 8 Sign extension bit

0 = sign extension required

1 = no sign extension required

RR - bits 9..10 Register number

00 - no register

01 - register 1

		10 - register 2
		11 - register 3
1	- bit 11	Indirect bit
		0 = non-indirect memory operand
		1 = indirect memory operand or operand not in memory
F	- bit 12	F bit in SEL addressing
N	- bit 16	Sign of register operands
Displacement	-	Bits 16..31 displacement in address calculations or literal values
Bits 13..31	-	Absolute addressing

The location of an operand is determined at run-time as follows: Bits 8 and 11 specify the location and sign of the operand according to the following encoding:

bit 8	bit 11	
0	0	Operand is signed in memory and not indirectly addressed
0	1	Operand is in memory and indirectly addressed
1	0	Operand is unsigned and in memory and not indirectly addressed
1	1	Operand is an address in a register (signed or unsigned) or a literal

If the operand is in memory the address is obtained using bits 9 and 10 and bits 12 through 31. Bits 9 and 10 specify the base register and bits 13-31 specify a displacement to be added to the base in computing the address. A base register of zero indicates that bits 13-31 contain the absolute address of the operand. Bit 12 is the F bit as in the SEL machine instructions and is used to specify 32 bit word, 16 bit halfword or 8 bit byte operands.

If the operand is not in memory, that is both bits 8 and 11 are set, and bit 9 is not set then the operand is an address or a literal. If bit 10 is not set the operand is a literal and its value is in bits 16-31 with bit 16 being the sign bit. If bit 10 is set the operand is an address and is computed as above.

If bit 9 is set, the operand is in a register specified by bits 9 and 10 (register 2 or 3) and bit 16 indicates its sign. By default, all addresses stored in memory begin on a word boundary, and the F bit is therefore not necessary when specifying indirectly addressed operands. Instead, for indirectly addressed operands the F bit is used to indicate that sign extension is to be done according to the following convention: 1 = sign extension required, 0 = no sign extension required. The flowchart in figure 5.2 shows this decoding algorithm.

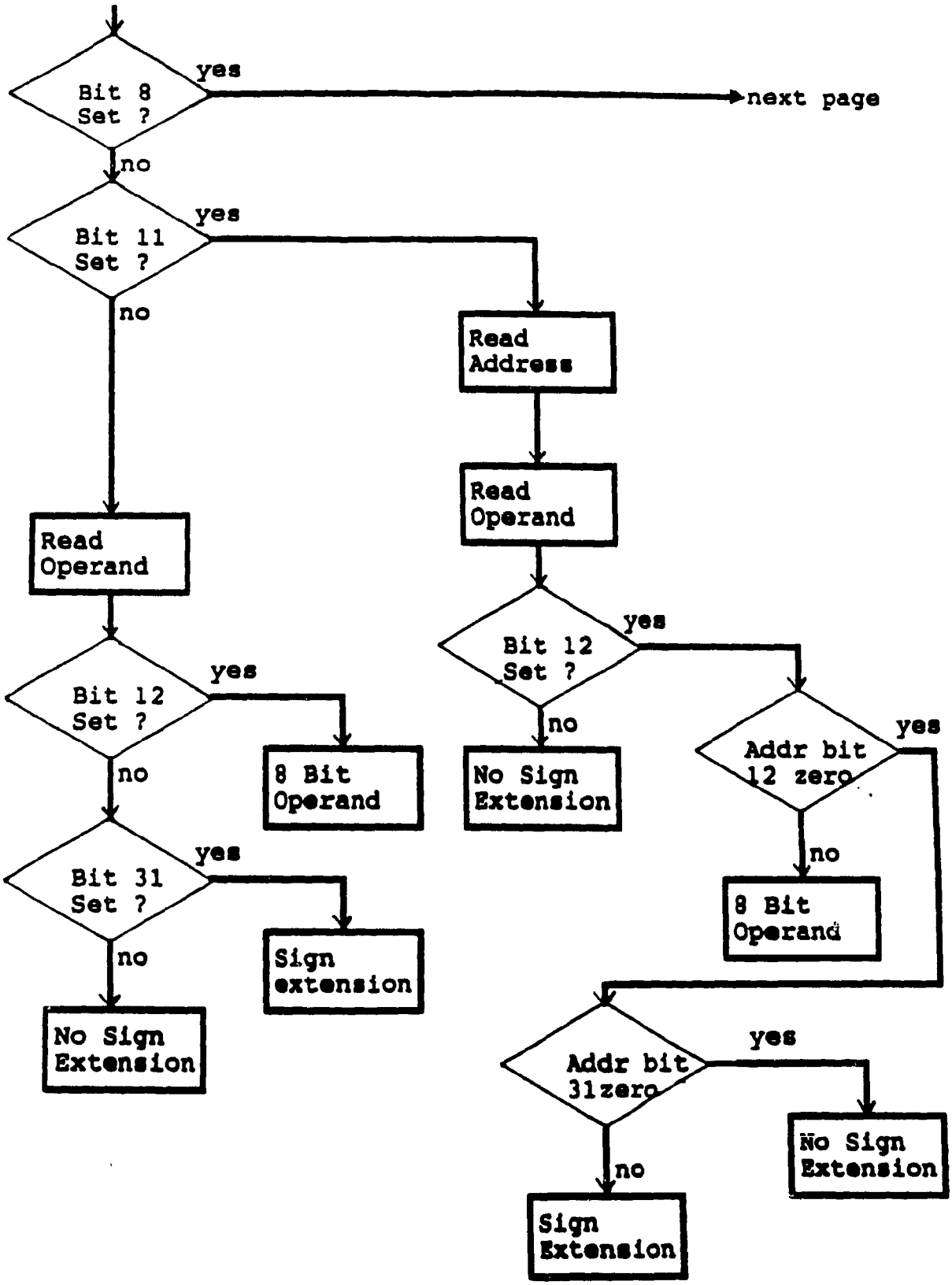


Figure 5.2: Operand Decode Algorithm (Page 1 of 2)

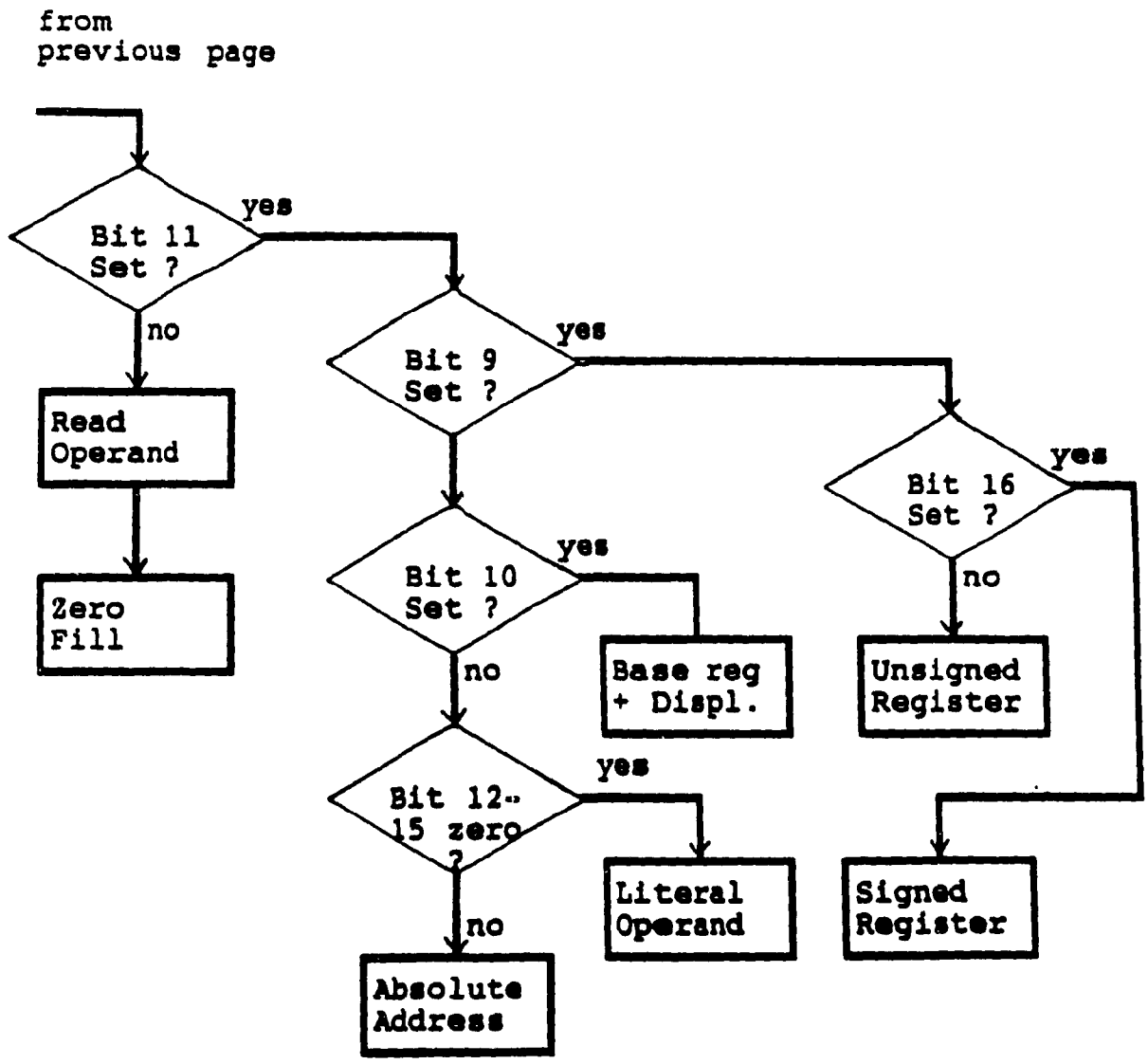


Figure 5.2: Operand Decode Algorithm (Page 2 of 2)

5.4 Design Considerations In Operand Specification

The format of Ecode-I was chosen to incorporate many of the features of the SEL microarchitecture and machine language. The following sections describe some of these features.

opcode - bits 0-7 - chosen for ease of decoding. The instruction can be stored into the T register and the T and S registers nibble shifted left together with the S acting as the most significant bits. The isolated opcode in the S register can be used as a jump table index.

sign extension bit - bit 8 - Chosen by default

Register operand or base register - bits 9 and 10. These bits are used to represent the index register in the SEL machine language. The microinstruction repertoire contains instructions which reference these two bits as register numbers. For example microinstruction MARIX adds the value of the base register indicated in bits 9 and 10 and the displacement in bits 12-31 of the instruction register and places the result in the memory address register. R(x) is a microinstruction

referencing the contents of the register indicated in bits 9 and 10.

Indirect bit - bit 11. This bit is used to indicate that an operand is indirectly addressed in memory, and is also used as the indirect bit in the SEL machine language. Microinstruction INDIR directly tests this bit when an address is placed in the memory address register, which facilitates easy checking to determine if a second memory read is necessary.

F bit - bit 12. This bit indicates the sign of indirect operands or the length (32, 16 or 8 bits) of non-indirect operands in memory. This is the addressing scheme is used in the SEL machine language and it takes advantage of the SEL memory reference logic which automatically reads the operands according to the value of this bit. Since this bit is not necessary when reading indirect operands, its use as a sign bit does not interfere with memory referencing.

Bit 16 - Indicates the sign of register operands. Register operands are indicated in bits 9 and 10 with the rest of the instruction unused. Bit 16

was chosen as the sign bit because it is easily tested by microinstruction BMUX00.

Bits 16-31 - Used as a literal or displacement. This is the same as in the SEL machine language. The microinstructions ZE and SE zero fills or sign extends this 16 bit field of the instruction to be used in arithmetic operations or to be stored.

Bits 12-31 - used as an absolute or relative address of an operand. This is the same convention as in the SEL machine language. The microinstruction MARIX loads these bits directly into the memory address register.

5.5 Sample Ecode-I Instructions

Below is the Ecode-I instruction generated for the instruction A = A + B, where B is a 16 bit signed integer on the run time stack and A is a 32 bit signed value in memory location at label U.

```
GEN   8/32,1/0,2/1,1/0,1/0,19/H(0)      opcode,operand B
GEN   8/0,1/1,2/0,1/0,1/0,19/W(U)       operand A
```

The representation of each field of this instruction is described below:

GEN	8/32, Opcode Add2 (32)	1/0, Sign bit zero. Sign ext. required	2/1, Base Reg R1	1/0, Indirect addr. off	1/0, F bit zero Oper. 32/16 bits	19/H(0) Displ. zero half words
GEN	8/0, Unused	1/1, Sign ext. not req.	2/0, No base register	1/0, Indirect addr. off	1/0, Oper 32/16 bits	19/W(U) Displ. word addr. of U

The Ecode-I representation of the CE procedure "push" in the module "stack" described in chapter three, and derived from its intermediate form also described in chapter three, is given below. The Ecode-I representation of the entire module is given in Appendix III.

```
  Ecode-I  Instruction                              Comment

PUSH  EQU    $
      GEN    8/25,1/1,2/0,1/1,1/0,19/1011          set line number
      GEN    8/81,1/0,2/0,1/0,1/0,19/H(U)          top = top + 1
      GEN    8/26,24/0                             inc. line  num
       GEN    8/32,1/0,2/0,1/0,1/0,19/H(U)         store top on
      GEN    8/0,1/1,2/1,1/0,1/0,19/W(0)           run time stack
      GEN    8/23,1/1,2/1,1/0,1/0,19/W(0)          shift top of
      GEN    8/0,1/1,2/0,1/1,1/0,19/1              stack left 1bit
      GEN    8/80,1/1,2/0,1/1,1/0,19/W(U)+4        set address to
```

```
GEN     8/0,1/1,2/1,1/0,1/0,19/W(0)     table + top*2
GEN     8/32,1/0,2/1,1/0,1/0,19/W(-8)   table(top)=I
GEN     8/0,1/1,2/1,1/1,1/0,19/W(0)
GEN     8/29,1/0,2/0,1/0,1/0,19/W(0)    return
```

5.6 Comparison Of SEL Machine Language And Ecode-I

Table 5.3 compares the features of the SEL machine language with Ecode-I. The major difference between the SEL machine language and Ecode-I is that the SEL opcode specifies the location and size of both operands, whereas Ecode-I opcode specifies neither.

Item	SEL	Ecode-I
Number of instructions	189	253
Opcode information	Opcodes specify operation, size, and location of operands.	Opcodes specify operation but not size and location of operands.
Instruction format	Opcode followed by operands. One operand is generally in a register.	Opcode followed by operands. Dest . operand specified last. Operand location unspecified
Number of operands per instruction	Zero, one or two.	Zero, one, two or three.
Length of instruction	1/2 - 1 word	1 - 3 words

Instruction decode		
Opcode	bits 0 - 5	bits 0 - 7
Source register	bits 6 - 8	bits 9 - 10
Index register	bits 9 - 10	bits 9 - 10
Dest. register	bits 9 - 10	bits 9 - 10
Indirect bit	bit 11	bit 11
F bit memory addressing	bit 12	bit 12
C bit memory addressing	bits 30 - 31	bits 30 - 31
Sign extension bit	not required	bit 8, or bit 16

Table 5.3: Comparison of SEL Machine Language and Ecode-I

5.7 CE Intermediate Code And Ecode-I

Ecode-I is derived from the CE intermediate code described in chapter 3. Each operation token in the intermediate code is mapped to an Ecode-I instruction. Additional Ecode-I instructions may be generated to calculate operand address based on the information in the datadescriptor tokens. For example, the intermediate code of the CE statement "I = table(top) " in the procedure "push." includes datadescriptors for table, I and subscript top. This generates three Ecode-I instructions to calculate the address of table(top) and one to perform the assignment. The translation of the intermediate code of procedure "push" to Ecode-I is shown below.

Intermediate Representation								Ecode-I Instruction
aRoutineIndex 0								
aIdentText 4 push								
aNewline 9								1. newline 9
aBegin								
aNewline 10								2. Incr line num
aDataDescriptor	162	1	0	0	2	0	0	3. Short add21:
aAssign								top
aDataDescriptor	162	1	0	0	2	0	0	
aDataDescriptor	1	127	1	0	1	0	1	
aAdd								
aEndExpression								
aNewLine 11								4. Incr line num
aDataDescriptor	162	1	3	0	20	0	2	5. Assign: top,
aSubs								top of stack
aDataDescriptor	162	1	0	0	2	0	0	6. Shift left:1,
aEndExpression								top of stack
aEndSubs								7. Add2: table
aDataDescriptor	1	127	1	0	1	0	1	address, top
aDataDescriptor	1	127	1	0	1	0	1	of stack
aDataDescriptor	1	127	1	0	1	0	9	8. Assign:
aDataDescriptor	129	127	0	0	2	0	0	indirect oper
aAssign								on top of
aDataDescriptor	162	2	0	0	2	-1	-4	stack, I
aEndExpression								

```
aNewLine  12                                9. Incr line num
aEndBegin
aNewLine  13                                10. incr line num
```

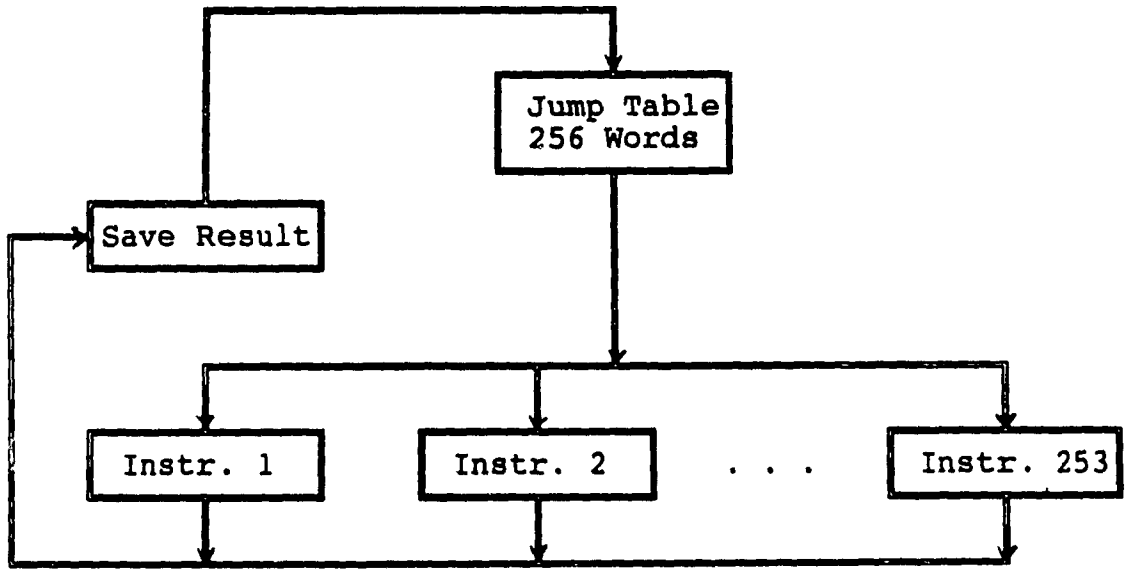
CHAPTER 6: DESIGN OF INTERPRETER I

6.0 Introduction

The intermediate code of the CE compiler is translated into Ecode-I for interpretation by the microprogrammed interpreter. The microinterpreter consists of a machine language main routine, a number of microcoded routines and the machine language coded I/O and predefined routines package used by the CE compiler. The total size of the interpreter is 1170 microwords of which 256 is a jump table. The microprogram consists of a jump table, a main program and five microsubroutines to fetch operands and addresses, as shown in figure 6.1.

An Ecode-I program is a SEL machine language program with the first instruction being a jump to the interpreter and the rest of the program consisting of data statements defining instructions to be interpreted. The machine language main routine establishes the processing environment and passes control to the microcoded interpreter in the writable control store. Certain functions such as I/O routines, multiply, divide and modulo instructions are implemented in machine language, mainly because they are complicated and require substantial microprogram memory to be implemented in the control store. Control is passed from

the microcoded routines to the machine language routines, and then back into the microprogram to continue interpretation. The microprogram execution environment is saved and restored each time the microprogram is exited or re-entered. Figure 6.2 shows the logical structure of the interpreter.



Operand fetch subroutines (called from Instr. i)

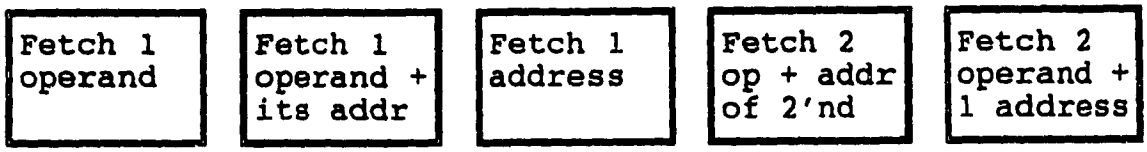


Figure 6.1: Implementation of Interpreter I

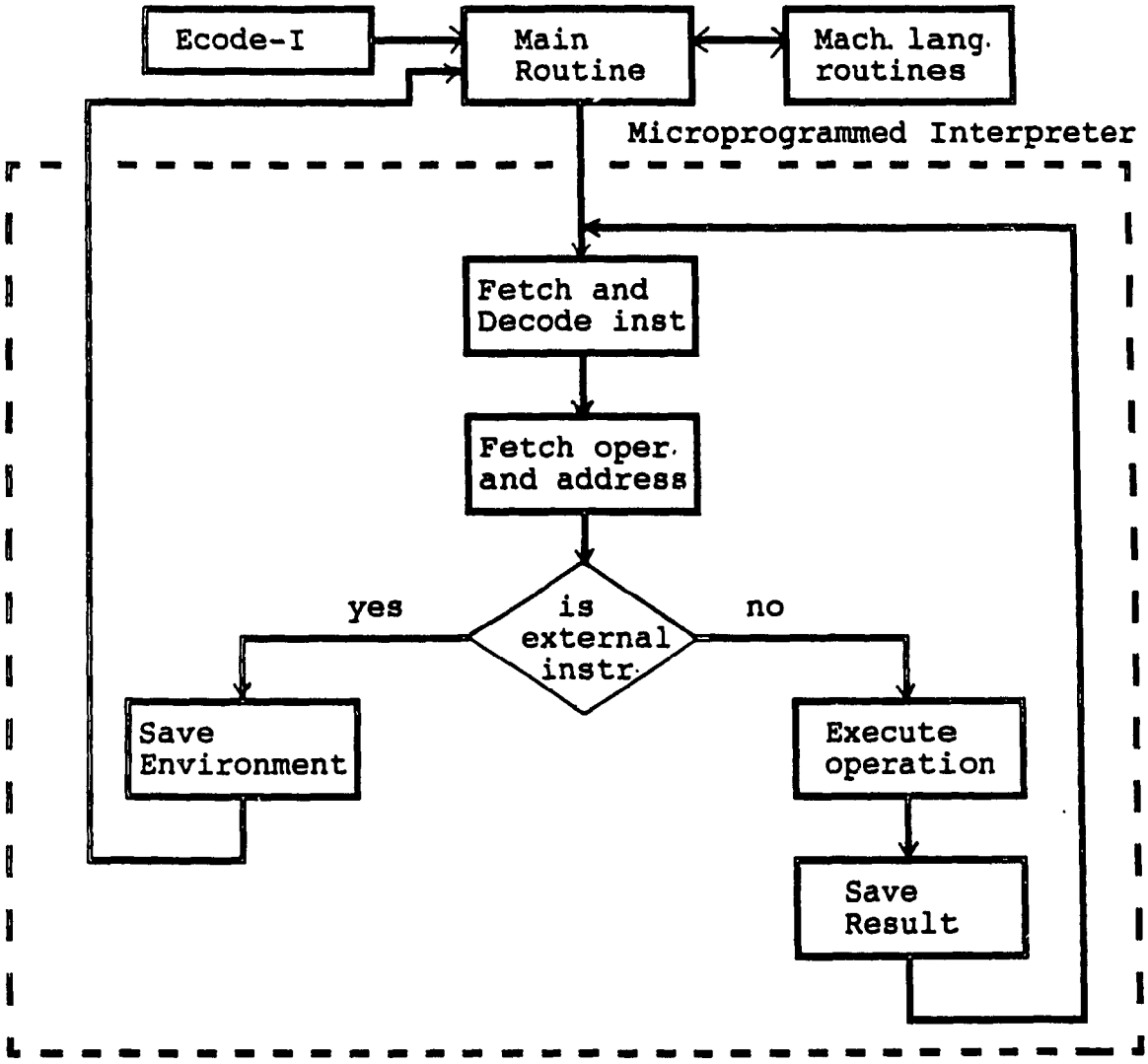


Figure 6.2: Logical Structure of Interpreter I

6.1 Parallel Execution

The memory cycle time on the SEL 32/75 is 900 nanoseconds which provides for the execution of six microinstructions during a memory fetch without any degradation of the execution time of the microprogram. This execution overlap is used extensively in the interpreter to maximize parallel execution of memory reads and instruction execution, and is illustrated in the following microprogram segment used in the interpreter. The Memory Read D and I columns indicate the elapsed time since the initiation of a data read and instruction fetch respectively, in units of 150 nanoseconds.

```
                                    Elapsed  Memory
                                      Time   Read
   Microinstructions                        D I  comment

   READ,CLRS;                          1    1 0  initiate read D
   NOD=@00800000&I0;                   2    2 0  Test sign bit
   IF INDIR *GOTO INDIR1,I1TOI0,NOD=S,SAVESIGN; is D indir
                                       3    3 0
   IF ALUZ *GOTO $+2,FETCHPC;          4    4 1  read next instr
   MARIX=R(X)+I0,SDEST,*GOTO C1;       5    5 2  address of A
            .
            .
C1 NOD=SIGN&I0;                        6    6 3  test sign bit
```

```
R(OP2)=DI(SE),READ;          7   0 4   store D in reg.
                                       read next
                                       operand
```

The interpreter was designed to maximize parallel execution of operations within a single microinstruction and memory reads. In general, fetching of the next instruction is overlapped with decoding the address of the operands and execution of the current operation. Where possible, decoding of the next instruction is overlapped with fetching the operands, executing the operation and storing the result of the present instruction. The least overlap occurs with a register operand as there is no memory data fetch to overlap the instruction fetch and decode with.

As a measurement of the degree of instruction execution overlap with memory reads, consider the actual microcode executed for sample instructions "A = B + C" and "A = A + D" where A, C, and D are in memory and B is a register. The actual microcode sections of the interpreter executed are shown in Appendix III, and the results duplicated below.

Expressions	Mcode I L1	P1	P1/L1 (overlap)
A = B + C	53	24	45%
A = A + D	39	21	54%

where

A = 16 bit signed integer on run time stack directly addressed

B = 16 bit signed integer in register three

C = 32 bit signed integer in global storage indirectly addressed

D = 16 bit signed integer on run time stack directly addressed

L1= number of microinstructions executed

P1= number of microinstructions executed in parallel with a memory read.

These results show approximately a 50% overlap of instruction execution with memory reads, and as expected, the degree of parallel execution is less when an operand is in a register.

6.2 Analysis And Benchmarking

The interpreter was benchmarked using the CPU bound prime number algorithm Sieve Of Eratosthenes shown in Appendix I. The results are summarized below.

algorithm	relative compiler time	relative interpreter time
Sieve of Eratosthenes	1	2.23

As shown the interpreter was unable to realize the expected performance gains. An analysis of the design of Ecode-I and the implementation of the interpreter suggested that the primary reason for this result was a lack of parallelism in instruction execution. It was observed that the least overlap in execution occurs during the instruction decode for instructions with a register destination operand, and that most of the execution time was spent determining the location of the operand. This is illustrated in Appendix III where for the evaluation of the expression "A = B + C" no overlap between memory read and instruction execution occurs until the thirteenth microinstruction in a total of fifty three instructions executed. Most of these thirteen instructions were used to determine whether operand B is in a register, in memory, an address, or in the instruction itself. To improve the execution speed of the interpreter a second interpreter was designed and implemented to reduce the instruction decode and operand address calculation time. This is described in the next chapter.

CHAPTER 7: ECODE-II AND INTERPRETER II

7.0 Instruction Set

The Ecode-I translator was modified to generate Ecode-II, designed to improve instruction decode and operand address calculation efficiency. The functional classification and number of instructions in the instruction set of Ecode-II is shown in table 7.1 below.

Classification	Number
Branch	5
Compare	15
Fixed Point Arithmetic	60
Logical	40
Set Manipulation	20
Short Fixed Point Arithmetic	60
Shift	2
Miscellaneous	14
Total	216

Table 7.1: Ecode-II Instructions by Category

Each instruction consists of an opcode followed by one or more operands. The opcodes were chosen to indicate the location of one or two operands depending on the number of operands in the instruction. For example, the Ecode-II instruction "Add2 MR" indicates an addition of two operands where the destination operand is in memory and the source operand is in a register. This is also the general format of the SEL machine instructions where the opcode indicates

the size and location of the operands. Sample instructions are illustrated in table 7.2 and a complete list is given in Appendix IV.

Opcode	# of Operands	Comment
Branch not equal	1	Branch to specified address if result of previous compare is not equal.
Shift Register	1	Shift register operand right/left. The direction of shift is specified in the instruction.
Add2 RM	2	Add a register to a memory operand and store the result in the memory operand.
Subtract3 AI	3	Subtract an immediate operand from an address operand and store the result in a third, unspecified, operand.
Multiply2	2	Same as Ecode-I.
Set Difference2 MR	2	Difference of two sets, one in a register and the other in memory; store the result in the memory operand.

Table 7.2: Sample Ecode-II Instructions

7.1 Operand Type

Each operand could be one of the five different types listed

below:

- Register operand (R)
- Memory operand (M)
- Address (A)
- Immediate operand (I)
- Short operand embedded in the opcode (S)

For three operand instructions the opcode specifies the location of the two source operands, and the location of the destination operand is determined at run time. As a result of the opcode specifing the location of the operands, the number of opcodes required to define the generic operations of Ecode-I, such as "Add2", "Subtract3" etc. , increased eight fold for two operand instructions and twelve fold for three operand instructions. To illustrate this, the Ecode-II opcodes required to implement "Add2" and "Add3" opcodes of Ecode-I are given below.

Add2 RM	Add2 MM	Add3 RM	Add3 MM	Add3 AM
Add2 RR	Add2 MR	Add3 RR	Add3 MR	Add3 AR
Add2 RI	Add2 MI	Add3 RI	Add3 MI	Add3 AI
Add2 RA	Add2 MA	Add3 RA	Add3 MA	Add3 AA

Eight Ecode-II opcodes required for Add2 opcode in Ecode-I.

Twelve Ecode-II opcodes required for Add3 opcode inh Ecode-I.

For three operand instructions the operand sequence IM, IR, IA and II do not require additional opcodes because the order of the operands and the opcode could always be modified to fit one of the twelve defined formats, as shown below.

Subtract3 IM - translated to - Add3 MI1 (where I1 = -I)
Subtract3 IR - translated to - Add3 RI1
Subtract3 II - done at compile time
Subtract3 IA - translated to - Add3 AI1

In reflexive operations such as add, multiply, etc., the order of the operands can be simply reversed, instead of negating the immediate operand and changing the opcode.

The implementation of eight or twelve Ecode-II opcodes for each Ecode-I opcode requires substantially more control store than the corresponding single instruction in Ecode-I, and as a result the instruction set of Ecode-II was reduced to 216 instructions compared to 253 in Ecode-I. Since each given instruction is repeated a number of times depending on the location of the operands and the length of the opcode remained at 8 bits the short operand format was implemented only for the "Assign", "Add3", "Subtract3" and "Logical AND3" instructions, as these were assumed to be the most frequently used. The following short Ecode-I instructions

were, therefore, not implemented in Ecode-II: short minus assign, short add2, short subtract2, short multiply3, short multiply2, short AND2, short OR3, short OR2, short set difference3, and short set difference2.

7.2 Opcode Decode

To improve opcode decode efficiency an opcode lookahead feature was added to the instruction, in which the opcode of the next instruction was also specified in the destination operand of the present two operand instruction, or in the second source operand of three operand instructions. This 8-bit opcode occupies the previously unused opcode field in the destination operand of the current instruction.

The lookahead feature makes it possible to overlap the decoding of the next instruction opcode with the following: fetching the next instruction, fetching the destination operand of the current instruction, executing the current operation and storing the results into the destination operand. This large potential operlap of opcode decode with the execution of the current instruction minimizes the overhead to decode the opcode, especially in cases of register destination operands where instruction decode was costly. In the case of a branch instruction where the branch is taken this saving is not realized as the

instruction decode must wait until the next instruction is fetched; if the branch is not taken execution is unaffected. The Ecode-II instruction, shown below, for the expression "A = B + C", where A, B and C are in memory, illustrates this feature.

Ecode-II Instruction	Comment
GEN 8/100, Operand B	Opcode 100, and Operand B
GEN 8/154, Operand C	Opcode of next instr. and C
GEN 8/0, Operand A	No Opcode and Operand A
GEN 8/154, Operand D	Next Instruction

As shown above, the opcode of the next instruction is also given in the previously unused opcode field of the second operand, thereby facilitating its decode even before the next instruction is fetched. The SEL assembly directives generated for the operands A, B, C and D are not given.

7.3 Operand Specification

The operand specification is the same as in Ecode-I with the following main changes:

- In memory destination operands the opcode of the next sequential instruction is stored in bits 0..7 which

were previously unused. In register destination operands the opcode of the next instruction is stored in bits 24..31, which were previously unused. The latter is easier to decode because no shifting is involved and these bits can be directly loaded into the jump register.

. In three operand instructions the location of the destination operand is determined by bit zero of the word defining the operand as follows: if bit zero is set, the operand is in memory and its address is specified by the rest of the instruction, otherwise it is in a register.

. In instructions with both memory and register operands the memory operand is generated first to allow maximum overlap of operand fetch with the rest of the instruction execution. This is done regardless of which is the destination operand.

. In the shift register instruction the sign bit indicates left or right shift; the number of bits to be shifted is indicated by bits 11 through 15; bits 24..31 indicates the opcode of the next instruction.

7.4 Sample Ecode-II Instructions

Below is the Ecode-II instruction generated for the sequence "A = A + B", where B is a 16 bit signed integer on the run-time stack and A is a 32 bit signed value in the memory location at label U.

```
GEN   8/60,1/0,2/1,1/0,1/0,19/H(0)      opcode,operand B
GEN    8/44,1/1,2/0,1/0,1/0,19/W(U)      next opcode,
                                         operand A
```

This representation of each field is of this instruction is described below:

GEN	8/60, Opcode Add2 MM (60)	1/0, Sign bit zero. Sign ext. required	2/1, Base Reg R1	1/0, Indirect addr. off	1/0, F bit zero Oper. 32/16 bits	19/H(0) Displ. zero half words
GEN	8/44, Next opcode Assign MM (44)	1/1, Sign bit set. Sign ext. not required	2/0, No base register	1/0, Indirect addr. off	1/0, F bit zero Oper. 32/16 bits	19/W(U) Displ. word addr of U

The Ecode-II representation of the procedure "push" in the CE module "stack" described in chapter three, and derived from its intermediate form which is also described in chapter three, is given below. The complete representation is given in Appendix IV.

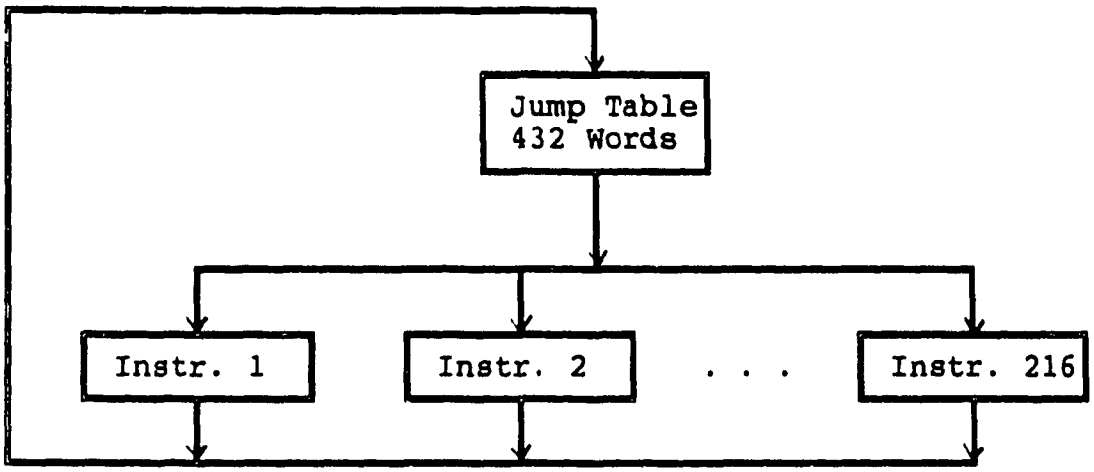
```
  Ecode-II Instruction                          Comment

PUSH    EQU     $
        GEN     8/32,1/1,2/0,1/1,1/0,19/1011  set line number
        GEN     8/154,1/0,2/0,1/0,1/0,19/H(U) top = top + 1
        GEN     8/33,1/0,2/0,1/0,1/0,19/1
        GEN     8/33,1/1,2/0,1/1,1/0,19/44    incr line number
        GEN     8/44,1/0,2/0,1/0,1/0,19/H(U)  stor top on stak
        GEN     8/31,1/1,2/1,1/0,1/0,19/W(0)  Shift top stack
        GEN     8/31,1/1,2/1,1/0,1/0,19/W(0)  left one bit
        GEN     8/0,1/0,2/1,1/0,1/0,5/1,16/59
        GEN     8/59,1/1,2/0,1/1,1/0,19/W(U)+4 set address to
        GEN     8/44,1/1,2/1,1/0,1/0,19/W(0)  table + top*2
        GEN     8/44,1/0,2/1,1/0,1/0,19/W(-8) table(top)=I
        GEN     8/36,1/1,2/1,1/1,1/0,19/W(0)
        GEN     8/36,1/0,2/0,1/0,1/0,19/W(0)  return
```

7.5 DESIGN OF INTERPRETER II

The logical structure of the interpreter, as shown in figure 7.1, has not changed much but its implementation is quite different. The interpreter consists of a main procedure with twenty six subroutines and is illustrated in figure 7.2. Each subroutine fetches operands in a specific order, for example, subroutine MR3 fetches two operands and one address, with the first operand in memory, the second in a

register, and the third either in memory or in a register. Unlike the previous interpreter there is no "pure" jump table; each entry of the jump table now occupies two microwords, instead of one, and initates operand address calculation in addition to ca ling a routine to fetch the operands and complete instruction processing.



Operand fetch subroutines (called from Instr. i)

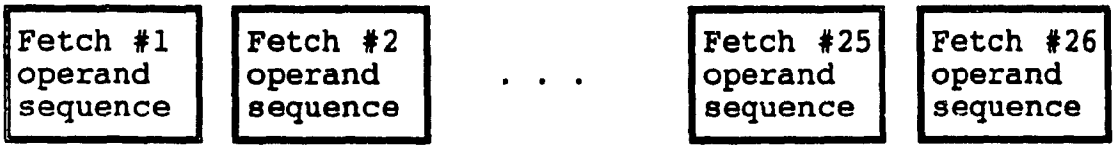


Figure 7.1: Implementation of Interpreter II

Also, unlike the previous interpreter in which results are assigned to destination operands in one centralized section of code, results are assigned to destination operands in the code that processes each instruction. Given the design of Ecode-II and the changes in the implementation of

interpreter II, the size of the interpreter has increased to 1893 microwords from 1170 microwords. Input and output operations, multiply, divide and modulo operations are still implemented in machine language and the same approach is used as in the previous interpreter.

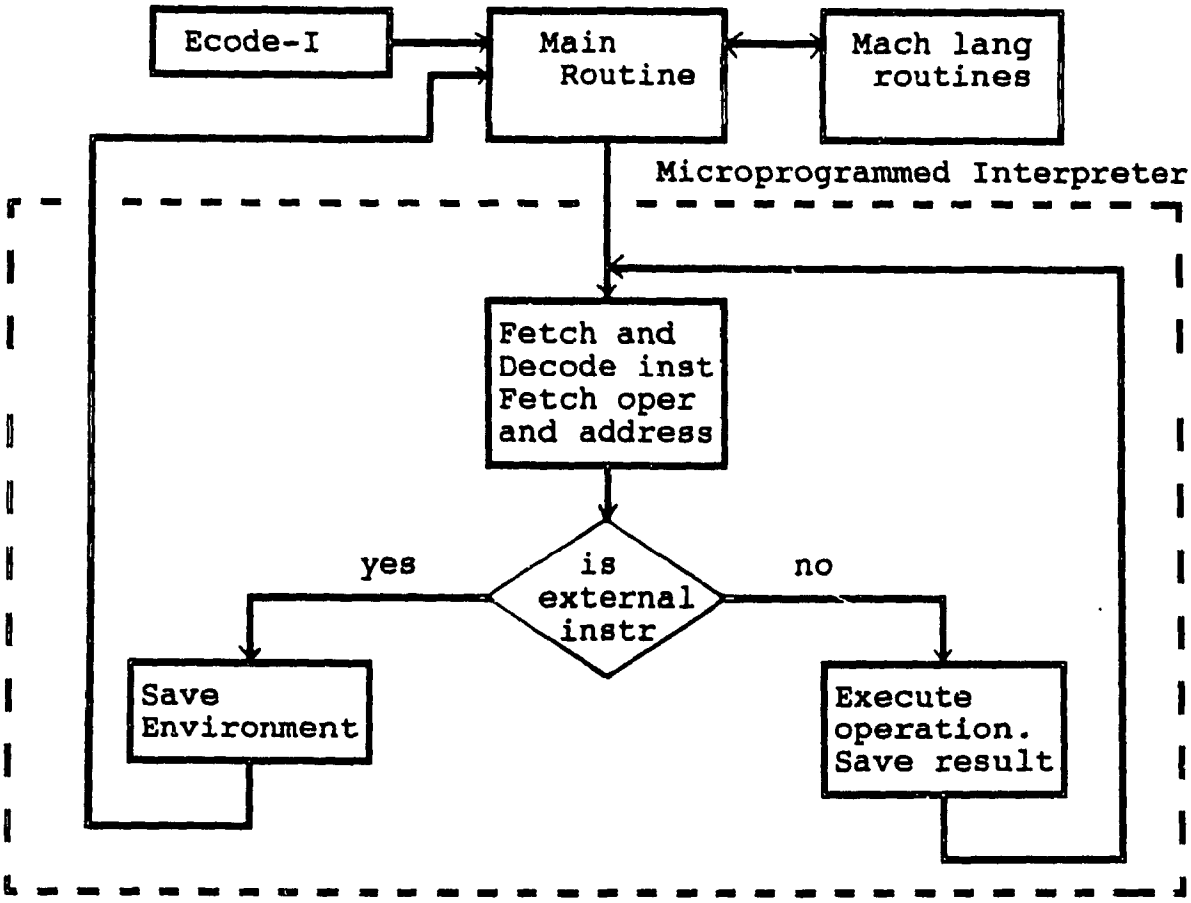


Figure 7.2: Logical Structure of Interpreter II

7.6 Parallel Execution

The design of Ecode-II facilitates parallel execution of microinstructions with data and instruction fetches. As in

the case of Interpreter I, the opportunity exists to overlap microinstruction execution with data and instruction reads. The following changes in Ecode and interpreter design facilitate parallel instruction execution and improves execution efficiency in Interpreter II.

1. Increasing the size of the jump table such that each instruction entry has two microwords instead of one. This permits address calculation and operand assignment to be initiated in parallel with a branch to the subroutine to execute the current or next instruction. This is possible only because the locations of the operands is specified in the opcode of the instruction.

2. There are twenty six operand fetch routines in interpreter II compared to five in Interpreter I. Each routine is tailored to read operands in a specific order and locations thereby enhancing its efficiency. This, again, is possible only because the locations of operands are specified in the opcode of Ecode-II.

3. Assignment of results to destination operands occurs "in line" with the code that performs the operations on the operands, as opposed to a single common block

of code in interpreter I. This permits optimization of parallel execution on a case by case basis as opposed to a general optimization.

As a measurement of the degree of instruction execution overlap with memory references, consider the actual code executed for sample instructions "A = B + C" and "A = A + D" where A, C and D are in memory and B is in a register, shown in Appendix IV. The results are compared with those of the previous interpreter and are duplicated below.

Expressions	Ecode-I L1	P1	P1/L1 (overlap)	Ecode-II L2	P2	P2/L2 (overlap)	L1/L2
A = B + C	53	24	45%	24	21	87%	2.24
A = A + D	39	21	54%	22	15	68%	1.77

where

A = 16 bit signed integer on run time stack directly addressed

B = 16 bit signed integer in register three

C = 32 bit signed integer in global storage indirectly addressed

D = 16 bit signed integer on run time stack directly addressed

L1, L2 = number of microinstructions executed

P1, P2 = number of microinstructions executed in parallel with a memory read.

As expected the degree of overlap of microcode execution with instruction and data fetches is substantially greater in Interpreter II, and the number of microinstructions executed by the new interpreter is almost half that of the previous interpreter.

7.7 Benchmarking

Benchmarking this interpreter with the same Sieve of Eratosthenes algorithm yields the results summarized below.

algorithm	relative time compiler	relative time interpreter I	relative time interpreter II
Sieve of Eratosthenes	1	2.23	1.12

These results are consistent with the observation above that the increase in speed is about twofold over interpreter I and comparable to that of the compiled code, and its significance and implications are described in the next chapter.

CHAPTER 8: ANALYSIS, FUTURE STUDY AND CONCLUSION

8.0 Analysis

The execution speed of the compiled code was measured to be faster than the interpretation of Ecode I and Ecode II. Analysis of the design of Ecode-I in Chapter 6 showed that its generic instruction set resulted in most of the execution time being spent in locating the operands and decoding the opcodes. Ecode II was then designed to more closely match the design of the SEL machine instructions in which the locations and sizes of the operands are specified by the opcode. Design changes in Ecode II and implementation improvements in Interpreter II resulted in a two fold increase in its execution speed over Ecode I. Given that the SEL micro-architecture is highly tuned to execute instructions in the SEL machine language format, Interpreter II is designed to take more advantage of SEL hardware support than Interpreter I does, to decode and execute the Ecode instructions.

The translation of the CE intermediate code to Ecode is relatively simple, compared to its translation into SEL machine language. Each Ecode instruction is the equivalent of one or more SEL machine language instruction, and as such it is not possible to generate Ecode in exactly the same

format as SEL machine language. Additionally, Ecode was designed for a stack architecture and the SEL is a non-stack machine, with little hardware support for stack management. The Ecode stack was implemented and managed by Ecode instructions. The resulting Ecode incompatibility with the SEL machine instruction format and SEL architecture increased the overhead of the Ecode programs, and reduced the SEL hardware support to the microprogrammed interpreters.

8.1 Future Study

In addition to modifying Ecode design to make better use of SEL hardware features, there are several other ways of improving the efficiency of Ecode interpretation. These include:

1. Modify the present interpreter design and implementation to improve the overlap between memory reads and microinstruction execution. As illustrated by the examples in Appendix IV the degree of parallel execution between microinstruction execution and memory reads is at least 50% (87% and 68% in the selected samples). The interpreter design could be modified to bring this figure closer to 100%.

Also the degree of overlap between instruction fetch and data fetch could be increased thereby creating the possibility for additional efficiency gains. In general, both of these possibilities would yield improvement in efficiency but would require an expert in SEL microcoding. This approach does not reduce the execution overhead; instead it addresses ways to improve execution efficiency.

2. Change the design of Ecode to include more powerful instructions which operate on groups of operands, thereby reducing the length of the Ecode program and therefore the number of instruction reads and decodes. On the other hand, more complex instructions are more difficult to decode. Since memory reads are so time consuming the reduced number of instructions may offset the increased instruction processing. Some examples of these instructions are:

   . Table B = 0 - Clearing the contents of an array could be achieved more efficiently as a tightly coded microcode loop as opposed to an Ecode loop.

   . Vector instructions such as ADD N, where N indicates the number of operands to add. For complicated expressions, these would

reduce the number of Ecode instructions generated.

3. Allocation of resources available to the microprogrammer directly in Ecode. In general the microprogrammer has many more hardware resources available than the machine language programmer. The length of the Ecode programs, instruction length and number of data reads from memory could be significantly reduced by allocation of some of these resources. Some examples in the SEL architecture are:

   . There are 256 scratchpad 32 bit registers on the SEL available to the microprogrammer and not included in Ecode. These could be used to partially implement the top of the run time stack where most indirect references to memory are made, and to store intermediate results. Additionally the length of the Ecode instructions would be reduced because addresses of these locations are 5 bits long instead of 24 bits.

   . There are 32 registers available to the microprogrammer, while only 8 are available to

the machine language programmer. Registers could be allocated to temporary variables, and used as predefined masks, etc.

. The memory address register, shift register, and program counter are all directly addressable to the microprogrammer and could be used as operands in Ecode.

Each of these modifications has potential to yield an improvement in the execution efficiency of the interpreter, however, the approach with the best promise for significant improvements is the one allocating resources available to the microprogrammer directly in Ecode.

An interesting extension of this project would be to implement the concurrency features into Ecode and the interpreter and to investigate its performance with the compiled version.

8.2 Conclusions

This study has been able to demonstrate, in a limited way, that a microcoded interpreter for Euclid will yield comparable results to the compiled code, and given certain changes in design and hardware support, has potential to yield substantial improvements in execution efficiency. The

importance of designing intermediate codes to match the host computer architecture and machine language design, in order to maximize hardware support for intermediate code execution has been established.

The project has also made it clear that the current emphasis of developing microprograming tools and high level languages for microprogramming, in the academic community, is an appropriate one. The tools available on the SEL for microprogramming development were a microprogram assembler and a microprogram loader, both of which were extensively used in the development of the interpreter. The lack of more advanced tools, such as a simulator, made the job of debugging the interpreter extremely difficult and time consuming. The timing problems, expecially in gating data and addresses onto busses and into registers, were very difficult to trace, and undebugged microcode frequently halted the computer at address locations unfamiliar to the microprogrammer, at which the state of memory or registers often had no meaning. Debugging also proved to be a tricky and time consuming task as the computer had to be rebooted, microprograms reloaded, and environment restored each time the system "crashed".

With the increasing development and availability of high level languages for microprogramming, the development of

microprogrammed interpreters should become increasingly simplified. The interpreter designer should be able to experiment with different intermediate codes and interpreter designs much more easily, and a better understanding of microprogrammable interpreters should lead to better designs and more efficient implementations.

REFERENCES

AGRA76 Agrawala, A. K., Raucher, T. G. Foundations of Microprogramming. Academic Press Incorporated 1976 QA76.6 A35.

BROA75 Broadbent, J. K., "High Level Language Implementation Through Microprogramming". Microprogramming and Systems Architecture, Infotech State of the Art Report 23 1975.

BROC73 Broca, F. R., Mervin, R. E., "Direct Microprogrammed Execution of the Intermediate Text from a High Level Language Compiler". Proceedings ACM Sigplan Sigmicro Interface Meeting May 1973.

BROW81 Brown, P. J., Writing Interactive Compilers and Interpreters. John Wiley & Sons 1979. QA76.6 B773.

COOP80 Cooper, R. E. M., " The Direct Execution of Intermediate Languages on an Eclipse Computer". SIGMICRO March 1980.

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

DuBO86 DuBose, D. K., Fatakis, D. K., Tabak, D., "A Microcoded RISC". Proceedings 19'th Annual Microprogramming Workshop December 1986.

FAGI85 Fagin, B., Pratt, Y., Srini, V., Despain, A., "Compiling Prolog into Microcode: A Case Study Using the NCR/32-000". Proceedings 18'th Annual Microprograming Workshop December 1985.

FLYN80 Flynn, M. J., "Interpretation, Microprogramming, and the Control of a Computer". Introduction to Computer Architecture. SRA 1980, QA76.9.A73I57.

GEE86 Gee, J., Melvin, S. W., Patt, Y. N., "Implementation of Prolog via VAX 8600 Microcode". Proceedings 19'th Annual Microprogramming Workshop December 1986.

HOLT83 Holt, R., Concurrent Euclid, The UNIX System, and TUNIS. Addison-Wesley 1983 QA76.73 C64H64

HABI81 Habib, S., Yang, X., " The use of a Meta Assembler to Design an Mcode Interpreter of AMD2500 Chips". SIGMICRO December 1981.

HASS76 Hassitt, A., Lyon, L. E., "An APL Emulator on System/370". IBM Systems Journal Volume 15 Number 4

June 1976.

JENS75 Jenson, K., Wirth, N., Pascal User Manual and Report. Springler-Verlag 1975.

LINE82 Linares, J. A Comprehensive Support System for Microcode Generation. Master's Thesis, Department of Computer Science. Concordia University, Montreal, Quebec Canada.

MICK77 Mick, J. R., "Microprogramming for the Hardware Engineer". SIGMICRO June 1977.

OKUN87 Okuno, H. G., Osato, N., Takeuchi, I., "Firmware Approach to Fast Lisp Interpreter". SIGMICRO 1987.

PATT85 Patt, Y. N., Melvin, S. W., Hwu, W., Shebanow M. C., "Critical Issues Regarding HPS, A High Performance Microarchitecture". SIGMICRO December 1985.

PERS77 Person, M., "Design of a Microprogram Generator for the Varian V73". SIGMICRO December 1977.

RAUC80 Raucher, T. G., Adams, P. M., "Microprogramming: A Tutorial and Survey of Recent Developments". IEEE Transactions on Computers Volume C-29 Number 1 1980.

REIG72 Reigel, E. W., Faber, U., and Fisher, D., A., "The Interpreter - A Microprogrammable Building Block System". AFIPS Spring Joint Computer Conference Proceedings 1972.

ROSI69 Rosin, R. F., "Contemporary Concepts of Microprogramming and Emulation". Computing Surveys, Volume 1, Number 4, December 1969.

SCHA83 Schaefer, M. T., Pratt, Y. N., "Improving the Performance of UCSD Pascal via Microprogramming on the PDP 11/60". Annual SIGMICRO Congress, 1983.

SEL1 Reference Manual, SEL 32/75 Computer. Systems Engineering Laboratories Incorporated August 1976. Publication Number 301-320075-00.

SEL2 SEL 32/75 Series Writable Control Storage Users Manual. Systems Engineering Laboratories Incorporated February 1979. Publication Number 301-322344-000.

SHRI81 Shriver, B., Lewis, T., "Introduction". IEEE Transactions on Computers July 1981.

SIGM86 Proceedings 19'th Annual Microprogramming Workshop. December 1986.

SIGM85 Proceedings 18'th Annual Microprogramming Workshop, December 1985.

STEV64 Stevens, W. Y., "The Structure of System 360 Part II". Bell and Newell Computer Structures 1964.

TRAC85 Tracz, W. S., "Advances in Microcode Support Software". Proceedings 18'th Annual Microprogramming Workshop December 1985.

TUCK65 Tucker, S. G., "Emulation of Large Systems" ACM Communications December 1965.

WILK69 Wilkes, M., V., "The Growth of Interest in Microprogramming: A Literature Survey". Computing Surveys, September 1969.

WILK84 Wilkes, J. L., "Architecture of a VLSI Multiple ISA Emulator". SIGMICRO December 1984.

WILN72 Wilner, W., "Design of the Burroughs B1700". Fall Joint Computer Conference 1972.

WIRT84 Wirth, N., "Lilith: A Modula Machine". Byte, August 1984.

APPENDIX I: CONCURRENT EUCLID

The following sections were taken from [HOLT83], and provide a brief description of some of the sequential features of Concurrent Euclid. A complete description can be obtained from [HOLT83].

I.1 Basic Data Types

CE has the traditional basic data types of Pascal, except float and enumerated types. There are several ranges of integers to reflect hardware data types. These basic types are:

Name	Values	Allocation
ShortInt	0..255	byte
SignedInt	-32768..32767	16-bit
UnsignedInt	0..65535	16-bit
LongInt	signed integer	32-bit
Boolean	false..true	byte
Char	a character	byte
AddressType	integer	address size
Pointer	address	address size

I.2 Structured Data Types

CE inherits the structured types of Pascal, namely arrays, records and sets. The following are example declarations using these types.

. Arrays - these are vectors of elements

```
var a: array 1..10 of SignedInt
var str: packed array 1..5 of Char
var matrix: array 1..5 of array 1..5 of LongInt
```

Variable a is an array of 10 Signed elements. Variable str is a character string. Variable matrix is the equivalent of a two dimensional array.

. Records - these are equivalent to Pascal records.

```
var r:
   record
        var status: Boolean
        var count:  SignedInt
   end record
```

This example declares r to be a record with fields called status and count.

. Sets - these are essentially bit strings.

```
var s: Set of 0..2
```

Set variable s is a set of three bits which can be individually changed and inspected.

I.3 Literal Values

A literal is an object which denotes its own value. CE literals include:

- Integer literals, e.g., 921 and 4887678
- Boolean literals, e.g., true and false
- Character literals, e.g., $X and $y. All character literals are preceded by the dollar sign ($) character.
- String literals, e.g., 'this is a test'. All character literals must be enclosed in quotes.

I.4 Control Structures

These include the following:

. loop, end loop, exit

. if, then, else, elseif, endif

. case, otherwise, endcase

. begin, end

I.5 Type Converters

CE has strong type checking; this means that the compiler disallows unlikely combinations of types such as adding the integer 14 to the Boolean value true. To allow for less rigorous type checking, CE defines TypeConverters, which do not generate any code but allow the bit pattern representing a value to be considered to be a value of another type.

I.6 Sample CE Module

Below is an example of a CE implementation of a stack. Two operations Push and Pop are defined on a data structure called Table. Push adds an item to Table and Pop returns the most recently added item to Table. The initially block sets the number of items in Table to zero when execution begins.

```
var stack:
    module
    exports (push,pop)
    const depth := 1..10
    var top: 0..Depth
    var table: array 1..Depth of signedint
procedure push(i:signedint)=
    imports (var top, var table)
    begin
```

```
            top := top + 1
            table(top) := i
        end push
    procedure pop(var i:signedint) =
        imports (var top, var table)
        begin
            i := table(top)
            top := top - 1
        end pop
    initially
       imports (var top)
       begin
           top := 0
       end
    end module
```

I.7 CE Input/Output Package

The CE I/O package has four levels of I/O sophistication which can be selectively "included" in the compilation process. These levels are as follows:

- IO/1 - terminal standard input/output (GET and PUT)
- IO/2 - program arguments and sequential files (READ and WRITE)
- IO/3 - temporary and non argument sequential files

(ASSIGN and DEASSIGN)

. IO/4 - record, array storage IO; random access files

The I/O package is independently coded and implemented. All references to it in the compiler are made via generated procedure calls.

I.8 CE Intermediate Code

The CE compiler makes four passes over the source input and its intermediate forms. The first three of these, the parser, semantic analyzer and storage allocator are machine independent. The fourth, the code generator, is the only one that has to be changed to port the compiler to another machine. The intermediate code, which will be transformed and interpreted by the microprogrammed interpreter, is the output of the storage allocation pass.

The intermediate representation of the module stack is shown below.

```
aNewline   1
aNewfile   26 PUB:[KUARLALL.TEST]TEST2.E
aModule
aIdenttext  5 STACK
aNewline   7
```

```
aProcedure
aRoutineIndex  0
aIdenttext 4 PUSH
aNewline  9
aBegin
aNewline  10
aDataDescriptor  162    1  0  0   2   0   0
aAssign
aDataDescriptor  162    1  0  0   2   0   0
aDataDescriptor    1  127  1  0   1   0   1
aAdd
aEndExpression
aNewLine  11
aDataDescriptor  162    1  3  0  20   0   2
aSubs
aDataDescriptor  162    1  0  0   2   0   0
aEndExpression
aEndSubs
aDataDescriptor    1  127  1  0   1   0   1
aDataDescriptor    1  127  1  0   1   0   1
aDataDescriptor    1  127  1  0   1   0   9
aDataDescriptor  129  127  0  0   2   0   0
aAssign
aDataDescriptor  162    2  0  0   2  -1  -4
aEndExpression
aNewLine  12
```

```
aEndBegin
aNewLine  13
aProcedure
aRoutineIndex  1
aIdentText  3  POP
    .
  the representation of procedure POP is similar
    .
aNewline  19
aInitially
aRoutineIndex
aNewline  21
aBegin
aNewline  22
aDataDescriptor  162    1  0  0   2   0   0
aAssign
aDataDescriptor    1  127  1  0   1   0   0
aEndExpression
aNewline  23
aEndBegin
aNewline  24
aEndModule
aEndOfFile
```

An aNewline token refers to the source line number that generated the code following it. The aNewFile token

indicates the file from which the source program was read. The most complex structure in the intermediate language is the specification of data objects. A simple data object is represented by an "aDataDescriptor" token which has five fields as shown below:

Status	Base	Representation	Value	Displacement

The status field is a bit string with the following encoding:

bits	significance
0 -	direct addressing
1 -	indirect addressing
2 -	double indirect addressing
3 -	register operand
4 -	index register
5 -	operand has a lexic base
6 -	operand on four byte boundary
7 -	operand on two byte boundary
8 -	auto decrement mode is on
9 -	auto increment mode is on
10 -	operand is temporary variable
11 -	operand is on runtime stack and a temporary variable
12 -	operand has temporary base register
13 -	operand has temporary index register

The base field describes the location of the operand. There are five possibilities:

1 - in global read/write storage area
2 - on runtime stack - local to module storage
3 - in global read only storage area
4 - in a register
127 - immediate operand in DataDescriptor itself

The representation field indicates the type of operand and the value field indicates the sign of the operand. In the case of a nonscalar operand, such as an array, the value field indicates the total size of the operand. The operand can be one of the following.

. float
. double float
. nonscalar - table or array etc.
. signed long (4 bytes)
. signed word (2 bytes)
. unsigned byte
. unsigned word(2 bytes)

The displacement field gives the displacement of the operand from one of the bases described in the base field. In the case of an immediate operand this field contains the actual

value of the operand. The displacement and value fields are both 32 bits long and are implemented as two 16 bit values. The other three fields are 16 bits long.

An array element requires at least six data descriptors for its specification. These are:

- start location and size of array
- subscript
- lower bound
- upper bound-1
- size of item
- attributes of item (eg.signed/unsigned)

Arrays of more than one dimension will have additional subscript datadescriptors.

I.9 Sieve Of Erathosthenes

The sieve of Erathosthenes is a prime number generation algorithm and is heavily CPU bound. It was used to benchmark the microprogrammed interpreter. Below is the CE implementation of the Sieve of Eratosthenes algorithm for prime numbers; this implementation also outputs the value of the largest prime number and the number of prime numbers found.

```
var sieve:
    module
    include 'io%3'
    initally
      imports var (io)
      begin
        var flags: array 1..8192 of signedint
        register var i: signedint
        var j: signedint
        var k: signedint
        var count: signedint
        var iter: signedint
        var prime signedint
        iter := 1
        loop
          exit when iter :=101
          iter := iter + 1
          count := 0
          i := 1
          loop
            exit when I = 8192
            flags(i) := i
            I := I + 1
          end loop
          i := 1
          loop
```

```
            exit when i := 8192
            if flags(i) not = 0 then
               prime := i + i + 1
               count := count + 1
               k := i + prime
               if k <= 8191 then
                 j := k
                 loop
                   exit when j >= 8191
                   flags(j) := 0
                   j := j + prime
                 end loop
               end if
            end if
            i := i + 1
          end loop
        end loop
        io.Putint(prime,8)
        io.Putstring(' is the largest of $e')
        io.Putint (count,6)
      end
    end module
```

APPENDIX II: SEL 32/75 COMPUTER

The SEL 32/75 is a high speed, general purpose, digital computer system. It is designed for a variety of scientific, data acquisition and real time applications. A basic system includes a central processing unit, main memory subsystem, and microprogrammed I/O controllers.

The following sections provide amplifying data on the SEL computer. A complete description can be obtained from [SEL1] and [SEL2].

II.1 SEL Assembly Language Directives

The intermediate code of the CE compiler was translated into Ecode-I and Ecode-II for interpretation by the microprogrammed interpreter on the SEL. Ecode-I and Ecode-II programs are generated using the SEL assembly language directives. The first instruction in an Ecode program is a jump to the interpreter and the rest of the program consists of the GEN data statement defining instructions to be interpreted. A description of the SEL assembly language directives used in Ecode generation is given below:

Directive/Instruction		Comment
PROGRAM	NAME	Indicates start of assembly language program called NAME
EXT	LABEL	Externally referenced name LABEL
DEF	LABEL	Defines LABEL for external reference
BOUND	N	Forces the program counter to an N byte boundary; for example N = 4 indicates fullword boundary and N = 2 indicates halfword boundary.
BL	LABEL	Branch and link to LABEL
GEN	N/B	Define N bits of memory with value B; for example GEN 8/1,8/2,8/3,8/4 generates the bit configuration: 0000 0001 0000 0010 0000 0011 0000 0100
LABEL EQU	VALUE	Equals tag; equates LABEL with VALUE
RES	N	Reserves N bytes of memory
END	LABEL	Marks end of assembly language program and indicates LABEL as starting address of execution.

APPENDIX III: ECODE-I AND INTERPRETER I

The folowing sections provide more detailed information on Ecode-I and Interpreter I.

III.1 Instruction Set

The Ecode-I instruction set consists of 253 instructions which are listed at the end of Appendix III by category.

III.2 Ecode-I Representation Of CE Module

The Ecode-I representation of the CE module stack described in Appendix I, and derived from its intermediate form also described in Appendix I, is shown below with comments.

```
        PROGRAM
        EXT     INTERPRET
        DEF     STACK
        DEF     PUSH
        DEF     POP
        BOUND   4
START   EQU     $
        BL      INTERPRET                       call interpret
        GEN     8/13,1/0,2/1,1/0,1/0,19/W(0)    zero top stack
        GEN     8/27,1/1,2/0,1/1,1/0,19/4       incr Stack ptr
```

```
STACK   EQU     $
        GEN     8/25,1/1,2/0,1/1,1/0,19/1001    set line number
        GEN     8/1,1/1,2/0,1/1,1/0,19/W(I34)   branch to I34
PUSH    EQU     $
        GEN     8/25,1/1,2/0,1/1,1/0,19/1011    set line number
        GEN     8/81,1/0,2/0,1/0,1/0,19/H(U)    top = top + 1
        GEN     8/26,24/0                       incr line num
        GEN     8/32,1/0,2/0,1/0,1/0,19/H(U)    stack <-- top
        GEN     8/0,1/1,2/1,1/0,1/0,19/W(0)
        GEN     8/23,1/1,2/1,1/0,1/0,19/W(0)    shift left 1bit
        GEN     8/0,1/1,2/0,1/1,1/0,19/1
        GEN     8/80,1/1,2/0,1/1,1/0,19/W(U)+4  set address to
        GEN      8/0,1/1,2/1,1/0,1/0,19/W(0)     table + top*2

        GEN     8/32,1/0,2/1,1/0,1/0,19/W(-8)   table(top)=I
        GEN     8/0,1/1,2/1,1/1,1/0,19/W(0)
        GEN     8/29,1/0,2/0,1/0,1/0,19/W(0)    return
POP     EQU     $
        GEN     8/25,1/1,2/0,1/1,1/0,19/1022    set line number
        GEN     8/32,1/0,2/0,1/0,1/0,19/H(U)    stack <-- top
        GEN     8/0,1/1,2/1,1/0,1/0,19/W(0)
        GEN     8/23,1/1,2/1,1/0,1/0,19/W(0)    shift left 1bit
        GEN     8/0,1/1,2/0,1/1,1/0,19/1
        GEN     8/80,1/1,2/0,1/1,1/0,19/W(U)+4  set address to
        GEN     8/0,1/1,2/1,1/0,1/0,19/W(0)     table +  top*2
```

```
        GEN     8/32,1/1,2/1,1/1,1/0,19/W(0)    I=table(top)
        GEN     8/0,1/0,2/1,1/0,1/0,19/W(-8)
        GEN      8/26,24/0                       incr line num
        GEN     8/81,1/0,2/0,1/0,1/0,19/H(U)    top = top + 1
        GEN     8/29,1/0,2/0,1/0,1/0,19/W(0)    return
I34     EQU     $
        GEN     8/25,1/1,2/0,1/1,1/0,19/1028    set line number
        GEN     8/13,1/0,2/0,1/0,1/0,19/H(U)    top=0
        GEN     8/29,1/0,2/0,1/0,1/0,19/0       return
        BOUND   4
M       EQU     $
        BOUND   4
U       EQU     $
        RES     6W                              top, table
        END     START
```

III.3 Parallel Execution

As a measurement of the degree of instruction execution overlap with memory reads, consider the actual microcode executed for sample instructions "A = B + C" and "A = A + D" where A, C, and D are in memory and B is a register. The microcode sections of the interpreter executed are shown in the next two sections. The results are summarized below. These results were obtained by totalling the number of microprogram statements executed for the expression in the

Expressions	Mcode I L1	P1	P1/L1 (overlap)
A = B + C	53	24	45%
A = A + D	39	21	54%

where

A = 16 bit signed integer on run time stack directly addressed

B = 16 bit signed integer in register three

C = 32 bit signed integer in global storage indirectly addressed

D = 16 bit signed integer on run time stack directly addressed

L1= number of microinstructions executed

P1= number of microinstructions executed in parallel with a memory read.

III.3.1 Execution Of Expression A = A + D

The Ecode-I instruction generated for this expression is:

```
GEN     8/32,1/0,2/1,1/0,1/0,19/H(0)    OPCODE:OPERAND D
GEN     8/0,1/1,2/1,1/0,1/0,19/H(0)     OPERAND A
```

The code segments which are executed for this expression are shown below with timing and overlap figures. The time column indicates the order in which the instructions are executed and the time, in machine cycles, at execution. The memory read column indicates the time, in machine cycles, which has elapsed for outstanding memory instruction (I) and data (D) reads. Each memory read takes six machine cycles, or 900 nanoseconds, and all microinstructions executed during this period are performed in parallel with the read. There is one entry in this column for each outstanding memory read. The program segments show only the code actually executed; the code for branches not taken, for example, is not shown.

```
                                          Memory
Label   Microinstructions          Time  Reads    comments
                                         D I

S_MASK  EQU      @00800000;                      sign mask
DECODE  EQU      0;                              decode reg
R_MASK  EQU      @00400000;                      reg mask
PCMASK  EQU      8;                              pgm cnt reg
OP1     EQU      2;                              operand 1
OP2     EQU      3;                              operand 2

        *GOTO ADD2                  1    0 0    jump table
```

```
                .
                .
ADD2    EQU       $
        *LINK     FETCH_2;            2   0 0  call fetch_2
        T=R(OP1);                    35   0 0  save oper 1
        S=S+R(DECODE);               36   0 0  decode instr
        T=R(OP1)+T,*GOTO ASSIGN;     37   0 0  add D, A
                .
                .
ASSIGN  IF SIGNSAVE *GOTO REG,NOD=I0;38   0 0  is A reg ?
        WRITE;                       39   0 0  write  A mem
        *JUMPS;                      40   0 0  next instr
                .
                .
FETCH_2 S=@00080000&I0               3   0 0  save F bit D
        NU=S_MASK&I0;                4   0 0  test sign ext
        MARIX=R(X)+I0;               5   0 0  address D
        IF ALUZ *GOTO POS1;          6   0 0  D in reg ?
                .
                .
POS1    IF INDIR *GOTO C27;          7   0 0  D indirect ?
        READ;                        8   1 0  read D
        INCRN;                       9   2 0  set flag
        IF %HWORD *GOTO $+2;        10   3 0  test sign ext
        NL=S_MASK;                  11   4 0  set flag
        I1TOI0,FETCHPC;             12   5 1  read nextinst
```

```
        S=@00080000&I0;                 13   6 2  save F bit A
        NOD=S_MASK&I0;                  14   0 3  test sign ext
        MARIX=R(X)+I0;                  15   0 4  address of A
        IF ALUZ  *GOTO POS2,R(OP2)=R(X);          A in reg ?
                 .                      16   0 5
                 .
POS2    IF NCTR4 *GOTO C14;             17   0 5  zero fill D ?
        IF NCTR0 *GOTO C41;             18   0 6  sign ext D ?
                 .
                 .
C41     R(OP1)=DI(SE),*GOTO C14;        19   0 0  store D inreg
                 .
                 .
C14     IF INDIR *GOTO C25;             20   0 0  locate A
        READ,NU=R(PCMASK),I1TOI0;       21   1 0  read A
        T=S:MAR,INCRN;                  22   2 0  store  addr A

        IF %HWORD *GOTO $+2;            23   3 0  sign ext ?
        NL=S_MASK;                      24   4 0  set flag
        FETCHPC;                        25   5 1  read nextinst
        SCRATCH(2)=T;                   26   6 2  save addr A
        MAR=T;                          27   0 3  return addr
        CLRS,T=I0;                      28   0 4  decode instr
        S=SNIBL,TNIBL;                  29   0 5  decode instr
        S=SNIBL,TNIBL;                  30   0 6  decode instr
        NOD=R(PCMASK),SAVESIGN;         31   0 0
```

```
        IF NCTR0 *GOTO C42;          32   0 0  sign ext C ?
                 .
                 .
C42     R(OP2)=DI(SE);               33   0 0  store A inreg
        *JUMPJ;                      34   0 0  return
```

III.3.2 Execution of Expression A = B + C

The Ecode-I instruction generated for this expression is:

```
  GEN     8/44,1/1,2/3,1/1,1/0,19/0      OPCODE:OPERAND B
  GEN     8/0,1/0,2/0,1/1,1/1,19/W(U)    OPERAND C
  GEN     8/0,1/0,2/1,1/0,1/0,19/H(4)    OPERAND A
```

```
                                          Memory
                                            Read
Label    Microinstructions          Time  D I   comments

S_MASK EQU      @00800000;                    sign mask
DECODE EQU      0;                            decode reg
R_MASK EQU      @00400000;                    reg mask
PCMASK EQU      8;                            pgm cnt reg
OP1    EQU      2;                            operand 1
```

```
OP2     EQU         3;                                  operand 2

          *GOTO ADD3                      1    0 0  jump table
                  .
                  .
ADD3    EQU         $
        *LINK       FETCH_3;              2    0 0  call fetch_3
        T=R(OP2);                        48    0 0
        S=S+R(DECODE);                   49    0 0  decode instr
        T=R(OP1)+T,*GOTO ASSIGN;         50    0 0  add B, C
                  .
                  .
ASSIGN  IF SIGNSAVE *GOTO REG,NOD=I0;    51    0 0  is A reg ?
        WRITE;                           52    0 0  write A  mem
        *JUMPS;                          53    0 0  next instr
                  .
                  .
FETCH_3 S=@00080000;                      3    0 0  save F bit B
        NU=S_MASK&I0;                     4    0 0  test sign ext
        MARIX=R(X)+I0;                    5    0 0  address of B
        IF ALUZ *GOTO POS1;               6    0 0  B in reg ?
        IF INDIR *GOTO DIR1;              7    0 0  B direct ?
                  .
                  .
DIR1    NL=@00080000;                     8    0 0  set flag
        NOD=R_MASK&I0;                    9    0 0  test B addr ?
```

```
        R(OP1)=R(X)+I0(SE);              10   0 0  address of B
        IF ALUZ *GOTO C19,NOD=I0;        11   0 0  B in addr ?
        IF BMUX16 *GOTO C1,T=R(X);       12   0 0  check sign B?
                .
                .
C1      I1TOI0,R(OP1)=T;                 13   0 0  no sign ext
        FETCHPC,*GOTO C24;               14   0 1  read nextinst
                .
                .
C24     S=@00080000&I0;                  15   0 2  save F bit C
        NOD=S_MASK&I0;                   16   0 3  test sign ext
        MARIX=R(X)&I0;                   17   0 4  address of C
        IF ALUZ *GOTO POS2,R(OP2)=R(X);           is C in reg ?
                .                        18   0 5
                .
POS2     IF NCTR4 *GOTO C14;             19    0 6  zero fill B
                .
                .
C14     IF INDIR *GOTO C25;              20   0 0  is Cindirect?
                .
                .
C25     READ,FRCWORD;                    21   1 0  read addr C?
        NU=@00080000&I0;                 22   2 0  save F bit C
        I1TOI0,FETCHPC;                  23   3 1  read nextinst
        IF NALUZ *GOTO $+2;              24   4 2  sign extendC?
        INCRN;                           25   5 3  set flag
```

```
        MAR=DI;                          26   6 4  address of C
        READ;                            27   1 5  read C
        IF NCTRZ *GOTO $+3;              28   2 6  sign ext C ?
        IF %HWORD *GOTO $+2;             29   3 0  is C 16 bits?
        NL=S_MASK;                                 not executed
        S=@0008000&I0;                   30   4 0  save F bit A
        NOD=R_MASK&I0;                   31   5 0  is a reg ?
        MARIX=R(X)+I0,*GOTO D1;          32   6 0  address of A
                .
                .
D1      IF NALUZ *GOTO DIR3;             33   0 0  locate A
        IF INDIR *GOTO INDIR3;           34   0 0  is Aindirect?
        T=S:MAR;                         35   0 0  set up addr A
        I1TOI0,FETCHPC;                  36   0 1  read nextinst
        SCRATCH(2)=T;                    37   0 2  save T reg
        MAR=T;                           38   0 3  MAR = addr A
        T=I0,CLRS;                       39   0 4  decode instr
        S=SNIBL,TNIBL;                   40   0 5  decode instr
        S=SNIBL,TNIBL;                   41   0 6  decode instr
        NOD=R(PCMASK),SAVESIGN;          42   0 0  set flag
        IF NCTR4 *GOTO J2;               43   0 0  sign ext C ?
        IF %NCTR0 *GOTO $+3;             44   0 0  is C 32 bits?
        R(OP2)=DI(SE);                             not executed
        *JUMPJ;                                    not executed
        IF NCTRZ *GOTO J1;               45   0 0  zero fill C ?
        R(OP2)=DI;                       46   0 0  store C inreg
```

```
*JUMPJ;                          47    0 0  return
```

Ecode-I Instructions By Category

1. Branch Instructions	opcode
branch always	1
branch greater than	2
branch less than	3
branch equal to	4
branch greater than or equal to	5
branch less than or equal to	6
branch not equal	7
branch never	8
branch to predefined routine	9
2. Miscellaneous Instructions	
make long word	10
set byte address	11
boolean not	12
zero operand	13
compare	14
abort	15
convert to set	16
nonscalar assign	17
negate	18
new line	25
increment new line	26
adjust stack pointer	27
set half word address	28
return	29
3. Fixed Point Arithmetic	
divide assign 3	19
divide assign 2	20
mod assign 3	21
mod assign 2	22
assign	32
minus assign	48
add 3	64
add 2	80
subtract 3	96
subtract 2	112
multiply 3	128
multiply 2	144

4. Logical Instructions	Opcode
logical and 3	160
logical and 2	176
logical or 3	192
logical or 2	208
5. Set Manipulation Instructions	
set difference 3	224
set difference 2	240
6. Shift Instructions	
shift left	23
shift right	24

5. Short Arithmetic Instructions	
short assign (1-15)	33 - 47
short minus assign (1-15)	49 - 63
short add 3 (1-15)	65 - 79
short add 2 (1-15)	81 - 95
short subtract 3 (1-15)	97 - 111
short subtract 2 (1-15)	113 - 127
short multiply 3 (1-15)	129 - 143
short multiply 2	145 - 159
6. Short Logical Instructions	
short logical and 3 (1-15)	161 - 175
short logical and 2 (1-15)	177 - 191
short logical or 3 (1-15)	193 - 207
short logical or 2 (1-15)	209 - 223
7. Short Set Manipulation Instructions	
short set difference 3 (1-15)	225 - 239
short set difference 2 (1-15)	241 - 255

APPENDIX IV: ECODE-II AND INTERPRETER II

The folowing sections provide more detailed information on Ecode-II and Interpreter II.

IV.1 Instruction Set

The Ecode-II instruction set consists of 216 instructions which are listed at the end of Appendix IV by category.

IV.2 Ecode-II Representation of CE Module

The Ecode-II representation of the CE module stack described in Appendix I, and derived from its intermediate form also described in Appendix I, is shown below with comments.

```
        PROGRAM
        EXT     INTERPRET
        DEF     STACK
        DEF     PUSH
        DEF     POP
        BOUND   4
START   EQU     $
        BL      INTERPRET                       call interpret

        GEN     8/8,1/0,2/1,1/0,1/0,19/W(0)     zero top stack
```

```
        GEN     8/27,1/1,2/0,1/1,1/0,19/4       incr Stack ptr
STACK   EQU     $
        GEN     8/32,1/1,2/0,1/1,1/0,19/1001    set line number
        GEN     8/1,1/1,2/0,1/1,1/0,19/W(I34)   branch to I34
PUSH    EQU     $
        GEN     8/32,1/1,2/0,1/1,1/0,19/1011    set line number
        GEN     8/154,0,2/0,1/0,1/0,19/H(U)     top = top + 1
        GEN      8/33,1/1,2/0,1/1,1/0,19/44      incr line num
        GEN     8/44,1/0,2/0,1/0,1/0,19/H(U)    stack <-- top
        GEN     8/31,1/1,2/1,1/0,1/0,19/W(0)
        GEN     8/31,1/1,2/1,1/0,1/0,19/W(0)    shift left 1bit
        GEN     8/0,1/0,2/1,1/0,1/0,5/1,16/59
        GEN     8/59,1/1,2/0,1/1,1/0,19/W(U)+4  set address to
        GEN     8/44,1/1,2/1,1/0,1/0,19/W(0)    table + top*2
        GEN     8/44,1/0,2/1,1/0,1/0,19/W(-8)   table(top)=I
        GEN     8/36,1/1,2/1,1/1,1/0,19/W(0)
        GEN     8/36,1/0,2/0,1/0,1/0,19/W(0)    return
POP     EQU     $
        GEN     8/32,1/1,2/0,1/1,1/0,19/1022    set line number
        GEN     8/44,1/0,2/0,1/0,1/0,19/H(U)    stack <-- top
        GEN     8/31,1/1,2/1,1/0,1/0,19/W(0)
        GEN     8/31,1/1,2/1,1/0,1/0,19/W(0)    shift left 1bit
        GEN     8/59,1/0,2/1,1/0,5/1,16/0
        GEN     8/59,1/1,2/0,1/1,1/0,19/W(U)+4  set address to
        GEN     8/44,1/1,2/1,1/0,1/0,19/W(0)    table  + top*2
        GEN     8/44,1/1,2/1,1/1,1/0,19/W(0)    I=table(top)
```

```
        GEN     8/33,1/1,2/1,1/0,1/0,19/W(-8)
        GEN     8/33,1/1,2/0,1/1,1/0,19/58      incr line num
        GEN     8/58,1/0,2/0,1/0,1/0,19/H(U)    top = top + 1
        GEN     8/36,1/1,2/0,1/1,1/0,19/1
        GEN     8/36,1/0,2/0,1/0,1/0,19/W(0)    return
I34     EQU     $
        GEN     8/32,1/1,2/0,1/1,1/0,19/1028    set line number
        GEN     8/8,1/0,2/0,1/0,1/0,19/H(U)     top=0
        GEN     8/36,1/0,2/0,1/0,1/0,19/0       return
        BOUND   4
M       EQU     $
        BOUND   4
U       EQU     $
        RES      6W                              top, table
        END     START
```

IV.3 Parallel Execution

As a measurement of the degree of instruction execution overlap with memory reads, consider the actual microcode executed for sample instructions "A = B + C" and "A = A + D" where A, C, and D are in memory and B is in a register. The microcode sections of the interpreter executed are shown in the next two sections. The results are summarized below and compared with those obtained with Ecode-I and interpreter I. These results were obtained by totalling the number of

microprogram statements executed for the expression in the actual implementation of the interpreter.

Expressions	Ecode-I L1	P1	P1/L1 (overlap)	Ecode-II L2	P2	P2/L2 (overlap)	L1/L2
A = B + C	53	24	45%	24	21	87%	2.24
A = A + D	39	21	54%	22	15	68%	1.77

where

A = 16 bit signed integer on run time stack directly addressed

B = 16 bit signed integer in register three

C = 32 bit signed integer in global storage indirectly addressed

D = 16 bit signed integer on run time stack directly addressed

L1, L2 = number of microinstructions executed

P1, P2 = number of microinstructions executed in parallel with a memory read.

IV.3.1 Execution of Expression A = A + D

The Ecode-II instruction generated for this expression is:

```
GEN     8/44,1/0,2/1,1/0,1/0,19/H(0)    OPCODE:OPERAND D
GEN     8/XX,1/0,2/1,1/0,1/0,19/H(4)    OPERAND A
```

XX = Next opcode

The code segments which are executed for this expression are shown below with timing and overlap figures. The time column indicates the order in which the instructions are executed and the time, in machine cycles, at execution. The memory read column indicates the time, in machine cycles, which has elapsed for outstanding memory instruction (I) and data (D) reads. Each memory read takes six machine cycles, or 900 nanoseconds, and all microinstructions executed during this period are performed in parallel with the read. There is one entry in this column for each outstanding memory read. The program segments show only the code actually executed; the code for branches not taken, for example, are not shown.

```
                                                Memory
                                                Read
Label     Microinstructions              Time D I      comments

SIGN    EQU     @00800000;                             sign mask
DECODE  EQU     0;                                      decode reg
R_MASK  EQU     @00400000;                              reg mask
PCMASK  EQU     8;                                      pgm cnt reg
OP1     EQU     2;                                      operand 1
OP2     EQU     3;                                      operand 2
STMASK  EQU     9;                                      equate tag
```

```
ADD2MM   MARIX=R(X)+I0,SDEST,*LINK MM2;          jump table
                                       1   0 0
         *GOTO MADD2;                  18  0 0
                  .
                  .
MADD2    IF %SIGNSAVE *GOTO $+2,T=R(OP2)+DI;     sign ext?
                                       19  0 0
         T=R(OP2)+DI(SE);              20  0 0   sign ext A
         WRITE,*GOTO JS3               21  0 0   store A
                  .
                  .
JS3       *JUMPS;                      22  0 0   next instr
                  .
                  .
MM2      READ,CLRS;                    2   1 0   read D
         NOD=SIGN&I0;                  3   2 0   sign ext D ?
         IF INDIR *GOTO INDIR1,I1TOI0,NOD=S,SAVESIGN;
                                       4   3 0   is Dindirect?
         IF ALUZ *GOTO $+2,FETCHPC;    5   4 1   read nextinst
        MARIX=R(X)+I0,SDEST,*GOTO C1;  6   5 2   address of A
                  .
                  .
C1       NOD=SIGN&I0;                  7   6 3   test sign A?
         R(OP2)=DI(SE),READ;           8   0 4   store D inreg
         IF ALUZ *GOTO C4;             9   0 5
```

```
        NOD=R(STMASK),*GOTO C4,SAVESIGN;        locate A
                                        10   0 6
        IF INDIR *GOTO MM2.IND,T=I0;    11   0 0  A indirect ?
        I1TOI0,FETCHPC;                 12   0 1  read nextinst
        MAR=S;                          13   0 2  MAR = addr A
        CLRS;                           14   0 3  decode
        S=SNIBL,TNIBL;                  15   0 4  next
        S=SNIBL,TNIBL;                  16   0 5  instr
        S=SLEFT+R(DECODE),*JUMPJ;       17   0 6  return
```

IV.3.2 Execution OF Expression A = B + C

The Ecode-II program generated for this expression is:

```
  GEN  8/44,1/0,2/0,1/1,1/1,19/W(U)        OPCODE:OPERAND C
  GEN  8/0,1/1,2/3,1/1,1/0,3/0,1/1,15/55   OPERAND B
  GEN  1/1,7/0,1/0,2/1,1/0,1/0,19/H(4)     OPERAND A
```

```
                                        Memory
                                        Read
Label    Microinstructions              Time D I  comments

SIGN   EQU        @00800000;                     sign mask
DECODE EQU        0;                             decode reg
R_MASK EQU        @00400000;                     reg mask
PCMASK EQU        8;                             pgm cnt reg
```

```
OP1       EQU     2;                                  operand 1
OP2       EQU     3;                                  operand 2
STMASK    EQU     9;                                  equate tag

ADD3RM    MARIX=R(X)+I0,*LINK RM3;     1   0 0   jump table
          BMUX=N,*GOTO ADD3;          21   5 0

                    .
                    .
ADD3      T=R(OP2),IF %BMUX00 *GOTO MP3;          dest. R/M ?
                    .                  22   6 0
                    .
MP3        T=R(OP1)+T,WRITE,*GOTO JS; 23   0 0   B+C write A
                    .
                    .
JS        *JUMPS;                      24   0 0   next instr
                    .                                    .
                    .
RM3       READ;                         2   1 0   read C
          NU=SIGN&I0;                   3   2 0   sign ext B ?
          IF INDIR  *GOTO INDIR2,I1TOI0,FETCHPC; next instr
                    .                   4   2 1
                    .
INDIR2    IF ALUZ *GOTO $+2,NOD=R(PCMASK),SAVESIGN;
                                        5   4 2   set flag
          NOD=R(STMASK),SAVESIGN;       6   5 3   save sign ext
          S=R(DECODE):I0;               7   6 4   decode  instr
```

```
                IF BMUX16 *GOTO $+2,R(OP1)=R(X);        sign ext C?
                                          8    0 5
                R(OP1)=T(ZE);                           not executed
                MAR=DI;                   9    0 6      addr of C
                READ,R(TMP)=S,I1TOI0;     10   1 0      read C
                BMUX=I0;                  11   2 0      test dest
                FETCHPC,IF BMUX00 *GOTO RIMM;           next instr
                          .               12   3 1
                          .
RIMM            MARIX=R(X)+I0,SDEST,IF SIGNSAVEZ *GOTO $+2;
                                          13   4 2      addr of A
                R(OP2)=DI,*GOTO $+2;      14   5 2       wait for C
                                          15   6 3      store C
                R(OP2)=DI(SE);                          not executed
                IF INDIR *GOTO IND3;      16   0 4      A indirect ?
                I1TOI0,FETCHPC;           17   1 5      next instr.
                                          18   2 6      wait for mem
                MAR=S,DECRN;              19   3 0      save addr ofA
                S=R(TMP),*JUMPJ;          20   4 0      return
```

Ecode-II Instructions By Category

1. Branch Instructions	Opcode
jump direct address	1
jump memory address	2
branch to predefined routine	3
branch not equal	24
return	36
2. Compare Instructions	
compare RR	9
compare RI	10
compare RA	11
compare RM	12
compare MR	13
compare MI	14
compare MA	15
compare MM	16
compare AR	17
compare AI	18
compare AA	19
compare AM	20
compare IR	21
compare IA	22
compare IM	23
3. Miscellaneous Instructions	
make long word	4
set byte address	5
boolean not	6
zero register	7
zero memory	8
abort	25
convert to set	26
nonscalar assign	27
negate register	28
negate memory	29
newline	32
increment newline	33
adjust stack pointer	34
set halfword address	35

4. Shift Instructions	Opcode
shift register	30
shift memory	31
5. Fixed Point Arithmetic	
assign RR	37
assign RI	38
assign RA	39
assign RM	40
assign MR	41
assign MI	42
assign MA	43
assign MM	44
minus assign RR	45
minus assign RI	46
minus assign RA	47
minus assign RM	48
minus assign MR	49
minus assign MI	50
minus assign MA	51
minus assign MM	52
add2 RR	53
add2 RI	54
add2 RA	55
add2 RM	56
add2 MR	57
add2 MI	58
add2 MA	59
add2 MM	60
subtract2 RR	61
subtract2 RI	62
subtract2 RA	63
subtract2 RM	64
subtract2 MR	65
subtract2 MI	66
subtract2 MA	67
subtract2 MM	68
add3 RR	93
add3 RI	94
add3 RA	95
add3 RM	96

8. Short Fixed Point Arithmetic

R indicates register operand
M indicates memory operand
I indicates immediate operand
A indicates address operand