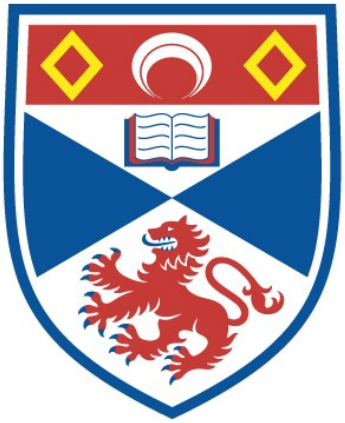
AN EXPERIMENT IN HIGH-LEVEL MICROPROGRAMMING

John F. Sommerville

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



1977

AN EXPERIMENT IN HIGH-LEVEL MICROPROGRAMMING

ABSTRACT

This thesis describes an experiment in developing a true high-level microprogramming language for the Burroughs B1700 series of computers.

Available languages for machine description both at a behavioral level and at a microprogramming level are compared and the conclusion drawn that none were suitable for our purpose and that it was necessary to develop a new language which we call SUILVEN.

SUILVEN is a true high-level language with no machine-dependent features. It permits the exact specification of the size of abstract machine data areas (via the BITS declaration) and allows the user to associate structure with these data areas (via the TEMPLATE declaration). SUILVEN only permits the use of structured control statements (if-then-else, while-do etc.) - the goto statement is not a feature of the language. SUILVEN is compiled into microcode for the B1700 range of machines. The compiler is written in SNOBOL4 and uses a top-down recursive descent analysis technique.

Using abstract machines for PASCAL and the locally developed SASL, SUILVEN was compared with other high and low level languages. The conclusions drawn from this comparison were as follows:-

(i) SUILVEN was perfectly adequate for describing simple S-machines

(ii) SUILVEN lacked certain features for describing higher-level machines

(iii) The needs of a machine description language and a microprogram implementation language are different and that it is unrealistic to attempt to combine these in a single language.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

AN EXPERIMENT IN HIGH-LEVEL MICROPROGRAMMING

JOHN F. SOMMERVILLE

This thesis descibes an experiment in developing and evaluating a high-level microprogramming language for the Burroughs B1700 series of computers.

The work on this project was carried out in the Department of Computational Science, University of St Andrews in fulfilment of the requirements for the degree of Doctor of Philosophy.

The study and research for this thesis has been carried out by myself and the thesis has been composed by myself.

The thesis has not been accepted in fulfilment of the requirements of any other degree or professional qualification.

ACKNOWLEDGEMENTS

Thanks are due to the staffs of the Deparment of Computational Science and the Computing Laboratory in the University of St Andrews, who provided help and encouragement during the course of this project. Special mention must be made of Mr R. Morrison, the project supervisor and Professor A.J. Cole.

CONTENTS

APPENDIX 1

A DESCRIPTION OF THE MICROPROGRAMMING LANGUAGE SUILVEN

APPENDIX 2

THE MICROARCHITECTURE OF THE B1700

APPENDIX 3

EXAMPLES

APPENDIX 4

AN ALGORITHM DESCRIPTION LANGUAGE

CHAPTER 1

INTRODUCTION

The research project described in this thesis originated from a dissatisfaction with the current state of programming language implementation, coupled with a belief that it was possible to improve that situation.

In particular, we were unhappy about the discrepancy between the machine-level facilities which we regard as necessary for the efficient implementation of high-level programming languages, and the actual facilities provided by most current machines. Therefore, the broad objective of our research was to study means of reducing this discrepancy by designing and constructing machines specifically geared towards executing high-level programming languages.

The material in this introductory chapter is split into three sections:-

(i) An overview of the general area of research.

(ii) A brief description of our particular research area.

(iii) An outline of the structure of the remainder of the thesis.

1.1 The General Area of Research

The development and implementation of high-level programming languages is essential to the successful application of computers to the solution of user-defined problems. Such languages must offer particular problem-solving facilities without imposing a severe cost overhead for the provision of these facilities. The general area of our research was to study techniques for achieving this aim.

Programming language implementation has been the subject of much research and, indeed, many of the problems associated with compilation have been solved. However, the implementation of a high-level language on most current computers is still a difficult and time-consuming task. Common difficulties arise in the implementation of both special-purpose and general-purpose programming languages.

These difficulties are largely due to the incompatibility between current machine architectures and the machine facilities required for the efficient implementation of high-level programming languages. In particular:-

(i) Artificial restrictions must be imposed on the high-level language to avoid heavy run-time penalties. For example, the overhead involved in run-time type checking, input-output, and dynamic storage allocation can be severe. Thus, powerful programming languages such as EULER, GEDANKEN, LISP, and SNOBOL become too inefficient for widespread use.

(ii) As current machine architectures are rarely oriented towards the execution of high-level programming languages, compilers must include extensive translation and optimisation facilities. The high-level language shields the programmer from the limitations of the machine, and he is often unaware of the space/time requirements of his executing code. This means that there is often no *a priori* method of determining the 'best' algorithm for solving a problem in a high-level language.

(iii) Generally, the machine architecture and instruction sets of machines marketed by different manufacturers are distinct. Hence, the implementation of a high-level language is usually 'once-off' and geared to a particular machine. Attempts to provide portable compilers can result in system inefficiencies or, the re-implementation of a compiler may require a programming effort which is almost as great as that involved in initially implementing that compiler.

The development of high-level languages has demonstrated that problem areas may usefully be categorised - numerical programming, list processing, text handling, etc, etc. It seems natural to develop separate programming languages specifically designed for encoding solutions to problems in each area and allow the user to tune this problem-oriented machine to his own application. Each of these problem-oriented languages should execute efficiently on a computer whose primitive operations and data types reflect the needs of its high-level language.

This language-oriented approach to machine design has been adopted by the Burroughs Corporation, initially in the B5000 range of machines described by Lonergan and King(L1), and more recently in the B5700/6700 machines, described by Organick(O1). These machines are oriented towards the efficient implementation of an extended ALGOL60, and include hardware features such as a stack, descriptor-based data organisation, and a reverse Polish instruction set.

However, the implementation of a language-oriented machine as a hard-wired unit, imposes restrictions on the implementation of programming languages, other than the language for which the machine is designed.

For example, the implementation of programming language compilers on B5700 computers is fraught with difficulty. Although the stack-based architecture suits many high-level languages, the machine instructions, data organisation, and conventions used by this machine are not necessarily suitable for other high-level languages. Adapting compilers to fit this strictly ALGOL-oriented organisation is extremely difficult.

Because of the restrictions imposed by a hard-wired machine geared towards a particular language, it is desirable that a language-oriented machine be available for executing each programming language. Presently, the only economically viable means of providing each language with its own machine is to emulate that machine using a computer program. Such a machine is known as an abstract machine, a virtual machine, a soft machine,

or an s-machine. These terms are used interchangeably throughout this thesis.

An early use of this language-oriented soft machine approach, was in the implementation of the Whetstone compiler for ALGOL60, described by Randell and Russell(R1). In this case, code was generated for an ALGOL-oriented machine called the Beta machine, and this was implemented via an interpreter. The Beta machine is an early example of a stack-oriented s-machine with an instruction set designed to efficiently implement ALGOL60. In particular, instructions for procedure entry and exit, and array bound checking are included.

Other implementations of language-oriented s-machines include Griswold's SNOBOL machine(G1,G3), and a machine designed for text handling applications, described by Poole et al(P1). Unlike the Beta machine, these machines were not implemented via an interpreter, but were bootstrapped onto a host machine using a macro processor to generate host machine code from the abstract machine code.

More recent soft machine implementations include the P-code machine for PASCAL, which is described by Jensen(J1), and a development of Landin's SECD machine(L2). This machine has been designed to efficiently execute a list-processing language called SASL, described by Turner(T1). Both the PASCAL and the SASL machine are interpretatively implemented, and are discussed in more detail in subsequent chapters of the thesis.

In designing a language-oriented soft machine there are three major factors which must be taken into consideration. These are:-

(i) The 'fit' between the soft machine and the high-level language.

(ii) The 'fit' between the soft machine and the underlying hard-wired machine.

(iii) The tools available for designing and implementing the soft machine.

An s-machine for a high-level language may be designed by incorporating instructions in that machine which directly implement the primitive operations of the high-level language. Examples of such operations might be procedure entry in ALGOL or record referencing in PASCAL. This approach avoids the generation of extra code by the compiler, reduces the demands made on the run-time system, and produces compact code. Hence the machine/language interface has been moved away from the hardware towards the programming language, and the required run-time interface is implemented via primitive s-machine operations.

The implementation of s-machines on conventional machines using an interpretative system, usually results in machines which run significantly slower than conventional machines. However, the technique of microprogramming, using a machine with writeable control store, can partially reduce this overhead. At little cost, s-machines may be defined to fit a high-level language and, by

allowing the control store of the microprogrammable machine to be shared, different s-machines may be multiprogrammed. A host 'hard' machine may support a number of different s-machines with each high-level language executing on a machine sympathetic to its requirements.

The broad objective of this project has been to develop tools for the design and implementation of s-machines. Up until now, s-machine design has been specified in an informal manner and, if the s-machine is microprogrammed, it has been encoded in a low-level language. We do not regard this situation as particularly satisfactory, as the implementation of s-machines using this technique is both time consuming and error prone. The specific aim of our project, therefore, was to develop tools for the documentation and implementation of s-machines.

1.2 The Particular Area of Research

The research project described here is a part of a larger research project at St Andrews University. This project is investigating methods of soft machine implementation and has as its ultimate aim, the automatic generation of s-machines from some formal specification.

The underlying microprogrammable machine which we chose to implement these s-machines was a Burroughs' B1726 computer. This machine, whose architecture is described in chapter 3, is designed explicitely for the emulation of abstract machines. In fact, there is no conventional machine code, and all languages

available on the system are implemented by translating them to an s-machine code and subsequently interpreting that code.

The B1726 is user microprogrammable and has a vertical microinstruction format. Control store may be dynamically reloaded, thus permitting the multiprogramming of microprograms. However, the most significant design feature of the B1700 is its bit-addressable store organisation.

The machine store is addressable to the single bit with no arbitrary word boundaries and no need for information alignment in store. Hence, the size of an information cell on this machine may be defined by the s-machine programmer and, indeed, may vary, depending on the machine instruction being interpreted.

This very flexible storage organisation coupled with the machine microprogrammability makes the B1700 eminently suitable for the implementation of s-machines. For this reason, the B1726 was chosen as the host machine for our project.

When the project was conceived(in 1973), it seemed likely that B1700 hardware would be available at St Andrews. Unfortunately, due to changed economic circumstances, our hopes were not fulfilled and no B1700 hardware became available locally. Naturally, this lack of hardware has been a significant constraint on our project.

As we were involved in the early stages of the larger research project, it was decided that a B1726 simulator should be constructed to act as an initial test-bed for microprograms whilst real hardware was unavailable. As events transpired, this simulator remains the only means of executing B1700 microprograms in St Andrews.

While the simulator was under construction, we considered the problems of designing and implementing s-machines.

We came to the conclusion that we should investigate the possibilities of microprogramming s-machines in a high-level language. We planned that this language(called SUILVEN) would not only be used for implementing s-machines but would alse serve as a vehicle for documenting and desribing s-machine architecture.

The advantages of programming in a high-level language are well known, but most microprograms are written in machine language. This technique is adopted for efficiency reasons - microprograms are frequently executed and should be as efficient as possible. Working in a research environment, there is no need to produce optimally efficient microprograms, and we wished to identify the benefits which accrue from high-level microprogramming. At the same time, we accepted the need for efficiency, and hoped that high-level microprograms would be comparable in efficiency with hand-coded, machine-level programs.

In order to investigate the efficacy of high-level microprogramming, we decided to compare SUILVEN implementations of two radically different s-machines, and to compare these implementations with implementations of these s-machines in other programming languages.

The s-machines chosen for this exercise were the P-machine for PASCAL(W1), described by Jensen(J1), and a development of Landin's SECD machine(L2), for the lambda-calculus based language SASL(T1). The P-machine is a fairly conventional stack machine, with a reverse

Polish instruction set. Instructions are included to implement special PASCAL operations such as set union, intersection, etc.

The SASL machine, on the other hand, is a higher-level machine with run-time type checking, a list-based storage organisation, and a high-level machine instruction set. As well as the usual reverse Polish operations, the SASL machine has a number of special-purpose instructions geared towards implementing unusual features of SASL.

To summarise therefore, the particular aims of this research project were as follows:-

(i) To program a simulator for the B1726 system at the microprogramming level. This simulator was to be implemented on our local IBM S/360 computer.

(ii) To design and implement a high-level microprogramming language for the B1700 range of computers. This language should also be suitable for describing s-machine architecture.

(iii) To evaluate the utility of this high-level language by comparing s-machine implementations in this language with the same machine implementations encoded in other programming languages.

1.3 The Structure of the Thesis

The remaining chapters of this thesis are devoted to a survey of the background to our particular research topic and to describing the research which we have undertaken.

Chapter 2 concentrates on the background to the design and implementation of SUILVEN. The chapter is split into four distinct sections:-

(i) A survey of high-level machine description languages.

(ii) An examination of the design philosophy behind the B1700 range of computers.

(iii) A description of microprogramming languages for the B1700.

(iv) A survey of current microprogramming languages for other user-microprogrammable machines.

Chapter 3 is a description of a B1726 simulator which was encoded in ALGOLW. We also describe the implementation of a compiler for MIL, the standard B1700 microprogramming language. These programs simulate the microprogramming environment on the B1700 and provide a means of implementing s-machines in a low-level programming language.

Chapter 4 is a description of the programming language SUILVEN. SUILVEN has a static storage allocation scheme, permits the exact specification of the size and structure of s-machine data areas, and allows the user to define procedures. The language control statements are simple and structured - the goto statement is not included.

Chapter 5 describes the implementation of the SUILVEN compiler. This compiler, written in SNOBOL4, is one-pass, uses a top-down recursive descent parsing technique, and generates a mnemonic form of B1700 microcode.

Chapter 6 is comprised of a general discussion of interpreter structure and a comparison of implementations of the PASCAL and SASL s-machines. The SUILVEN implementations are compared with implementations in PASCAL, PL360, BCPL, and MIL.

Chapter 7, the concluding chapter, discusses how well the project achieved its objectives and possible improvements which could be made to our system. We finally conclude that a separation of the functions of machine description and machine implementation is a better approach than that we adopted and suggest further research topics in this field.

There are a number of appendices which contain material rather too detailed to be included in the body of the thesis. These are:-

(i) A reference manual for SUILVEN.

(ii) A description of the B1700 microarchitecture.

(iii) Program listings of the PASCAL and SASL s-machines, plus test results from executing these machines.

(iv) A description of the notation used to describe algorithms throughout this thesis.

CHAPTER 2

BACKGROUND MATERIAL

This chapter consists of a survey and review of background material relevant to the research work described in this thesis.

Currently, virtually all work on microprogramming and abstract machine design has been of an engineering nature. Little or no theoretical background exists and the approach taken by research workers is to approach systems design on an 'ad hoc' basis. The resultant system is then tested and evaluated.

While this is a practical approach, it results in the building of special, one-off, systems. As a result, research in this field can only use these systems as a guide rather than as building blocks for further work.

This lack of theoretical background may be attributed to a number of factors:-

(i) Technological developments of system components has been extremely rapid. This new technology offers orders of magnitude improvements in speed and storage capacity. As a result, previously impractical systems may now be implemented and existing systems operate both more quickly and more cheaply. There has been little need to analyse and develop systems in order to produce improvements and little general knowledge of machine design has emerged.

(ii) Up until fairly recently, computer design was the province of the electronics engineer. The prime aim was to extract maximum hardware performance and little attention was paid to the requirements of the machine software.

(iii) Very little experimental work has been done regarding machine design for software systems. This is primarily due to the difficulty and expense of carrying out such research. To do so realistically,requires normal computer users to use an experimental system. Naturally they are reluctant to do so,and hence the machine designer must postulate a design on the basis of his own experience.

This lack of general knowledge is regrettable, but we cannot honestly envisage the situation changing in the near future. Paradoxically, because computing is so expensive, it is not possible to conduct some experiments which might ultimately reduce computing costs. To do so can result in an unacceptable increase in the immediate cost to the user.

The background material covered in this chapter is basically a study of existing microprogramming and machine description languages. In addition, as the B1700 computer is central to our project, we present an overview of the machine design and microprogramming languages developed for that machine.

This material is presented in four distinct sections:-

(i) Machine Description Languages

(ii) The Design of the B1700

(iii) Microprogramming the B1700

(iv) Microprogramming Languages for other Machines

2.1 Machine Description Languages

This section of the survey describes the uses of machine description languages, identifies various types of machine description languages, and briefly surveys several languages which have been used to describe machine architecture. In general, these languages have been used to describe 'hard' machines but the section concludes with a discussion on the description of soft machines.

Machine description languages were originally developed by digital engineers and their prime function is to document and hence communicate a machine design. A computer system may be described at a number of different levels and machine description languages have been designed to describe each of these levels. The levels in a system are:-

(i) The Algorithmic Level

This level is the logical description of the machine architecture. That is, it provides a definition of the machine data areas which are visible to an executing program and describes the instructions which operate on these data areas.

(ii) The Configuration Level

This level, often called the PMS(Processor, Memory, Switch) level, defines the interconnection between the various units in the system such as store modules, arithmetic function boxes, etc.

(iii) The Register Transfer Level

A machine description at this level describes how information is transferred between the system registers. This information flow executes machine instructions. The register transfer level is the level at which microprograms operate.

Below these levels there are, of course, levels concerned with machine electronics but these are not generally considered the province of the computer scientist.

The work covered in this dissertation is concerned with levels (i) and (iii), that is, the algorithmic level and the register transfer level. Register transfer level languages are discussed in sections 2.3 and 2.4.

Languages covered in this section have all been used to describe machine architecture. In general they have been specifically designed for this purpose although APL, the first language discussed, is a more general purpose language.

2.1.1 APL

The form of the language APL, designed by Iverson(I1) is now well known. The language is primarily suited to vector manipulation as it has special purpose operators for working with arrays.

Iverson(I2) maintains that APL may be used for describing digital systems both at an algorithmic and at a register transfer level. Indeed, he has used it to describe the architecture of the S/360 range of computers(I3). Iverson has categorised APL as a universal, concise, precise, systematic language which is easy to learn, to remember, and to use.

Be that as it may, we do not consider APL to be a particularly good machine description language for the following reasons:-

(i) The lack of declarations makes the description of machine data areas almost impossible.

(ii) The APL character set is not widely available.

(iii) The syntax of APL is unnatural and rebarbative, primarily because of its right to left expression evaluation. APL programs, although concise, tend to be unstructured and unreadable.

2.1.2 AHPL

AHPL(A Hardware Programming Language) is a development of APL by Hill(H1). He designed the language specifically to describe the design of digital systems.

The primary addition to APL is the introduction of declarations into the language, to permit the description of machine data areas.

For example:-

```
REGISTERS A(4),B(4),C(4)
```

This would define 3 4-bit registers A, B, and C.

However, there is no means of assigning any structure to these registers. The language, apart from its declarative facilities, has similiar operations to APL and consequently shares APL's disadvantages.

2.1.3 SFD-ALGOL

The language SFD-ALGOL, developed by Parnas(P2), was an early attempt to design a machine description language based on ALGOL 60.

SFD -ALGOL is essentially oriented towards describing 'hard' machines with the language extensions including declarations specifying which procedure parameters are input and output parameters, and 'time blocks'. The inclusion of input/output declarations allows a procedure to be regarded as a 'black box' accepting certain inputs and producing certain outputs. Time blocks specify which operations are carried out during a single system clock pulse.

Parnas did not introduce facilities into SFD-ALGOL for specifying the width or the structure of machine data areas and this is a serious drawback to its use as a machine description language. In general, the language is more suited to describing the operation

of a particular hardware component rather than describing the overall behaviour of a system.

2.1.4 APDL

APDL(Algorithmic Processor Description Language) is a development of SFD-ALGOL. The language was designed by Darringer(D1).

For providing a behavioural system description, the language is a considerable improvement over SFD-Algol. Darringer has introduced a new type into ALGOL called a binary register type. Using this the width of data areas may be specified. For example:-

(i) BINARY REGISTER ACC <0:23>

(ii) BINARY REGISTER ARRAY MEMORY [1:nopages,1:128]<0:31>

(i) above specifies that ACC is a 24-bit register while (ii) specifies that MEMORY is an array of 32-bit words split into pages. Each page contains 128 words. APDL also allows the user to associate some structure with declared registers by declaring subregisters. For example:-

BINARY SUBREGISTER FLAGS <1:2> = ACC<0:1>

This associates the name FLAGS with the first two bits of ACC.

The time blocks and input/output declarations from SFD-ALGOL have been retained in APDL as have the control constructs of ALGOL 60. As an example of APDL, a short segment of a program describing a minicomputer is shown below. The reader may assume that names used in the program have been previously declared.

```
cycle:  time begin
        IR := MEMORY [ MADR ] : MADR := MADR + 1
        go to  INSTRUCTION [ IR <2> + 1]
        end
cla:    time begin
        ACC := MEMORY [ IADR ]
        go to out
        end
add:    time begin
        ACC := ACC + MEMORY [ IADR ]
        go to out
        end
        .
        .
        .
stop:   print (0)
```

An Example of an APDL Program

The above program shows how APDL describes instruction fetching, the load accumulator instruction(cla) and the add instruction.

2.1.5 ISP

ISP, designed by Bell and Newell(B1) is another machine description language with an ALGOL-like structure.

Tbe language contains a number of features which make it a reasonably good machine description language.

(i) A means of specifying the width of machine data areas, plus facilities for ascribing structure to these areas.

(ii) A powerful conditional construct, similiar to the guarded statement command recently proposed by Dijkstra(D4). This conditional statement is very useful for selecting statements for execution using the instruction op code.

(iii) The usual logical and arithmetic operations plus a feature for specifying which operations may be executed in parallel.

The examples below, taken from the ISP description of the PDP-8 computer, illustrate the language

```
INSTRUCTION REGISTER\IR  <0:11>
```

This sets up a 12-bit register called IR, which acts as an instruction register.

```
OPERATION CODE \OP <0:3> := IR<0:2>
```

This specifies that the first three bits of IR may be regarded as the instruction op code.

```
( OP = 0 => AC := AC Λ MP [ EA ]  )
```

This statement illustrates the form of the ISP conditional statement.

Translated into the more familiar ALGOL this statement is:-

```
if OP = 0 then
    AC := AC and MP [ EA ]
```

These ISP conditional statements may be combined in a list as shown below

```
( OP = 0 => s1 )
( OP = 1 => s2 )
      .
      .
      .
( OP = 15 => s16 )
```

Effectively, a guarded statement list has been built with only the ISP statement preceded by a condition which is true being executed.

ISP, like the other languages described above, has been developed by digital engineers for describing 'hard' machines. These machines tend to have a simpler structure than language oriented soft machines, as 'hard' machines usually lack higher level machine operations. Although both APDL and ISP could be used for abstract machine descriptions their hardware oriented features would make such descriptions rather clumsy.

2.1.6 MDL

MDL(Machine Description Language) is the only language referenced in the literature which is specifically designed for describing abstract machines. It was designed by Wortman(W2) for describing the architecture of a PL/1 machine.

MDL, whose syntax is based on PL/1, is a high-level language whose features include procedures, if-then-else statements, while-loops, and case statements. Language declarations allow the width and structure of data areas to be specified exactly.

To illustrate MDL, the MDL code describing the PL/1 machine instruction CATENATE is shown below. The meaning of the MDL code should be obvious to the reader familiar with high-level languages.

```
CATENATE
        local p,q
        " procedure catenate uses a data stack called ds,
          with stack pointer dsp.  ds is structured into
          fields with the operand value held in field v
          and its length held in a subfield of v called a "
        " force types to char type, sum operand lengths
          and obtain space for result  "
        force_type(char,dsp,dsp-1)
        q ← ds[dsp].v.a + ds[dsp-1].v.a
        p ← string_space(q)
```

```
" copy operands to string store and create
  descriptor for result  "
move(ds[dsp-1].v,p)
move(ds[dsp].v,p+ds[dsp-1].v.a)
ds[dsp-1].v ← (q,p)
popds
```

An Example of an MDL Description

We considered using MDL as a vehicle for our experiments but finally rejected it because of language features such as compound statements being bracketed by indentation. Such features cause compilation problems as can certain aspects of the PL/1 based language syntax.

2.2 The Design of the B1700

As the B1700 series of computers was chosen as the implementation vehicle for our project, it is appropriate to present an overview of the design of that machine.

The B1700 architecture is a radical departure from conventional machine architectures as it is designed solely to support the emulation of language oriented abstract machines. The design philosophy is expounded in a series of papers by Wilner(W3,W4,W5) and the design is fully documented in the B1700 System Reference Manual(B2).

The fundamental design tenet of the B1700 is flexibility. This has been achieved by leaving certain machine features to be specified by the implementor of the abstract machine rather than by the hardware designers. These features are:-

(i) The size of the basic memory cell

(ii) The machine instruction set

Generally these features are fixed by the hardware engineers and this immediately imposes certain machine constraints. These are:-

(i) By fixing the machine cell size, operand precision is defined. In order to circumvent this, clumsy high-level language constructs such as double precision type variables must be introduced.

(ii) The implementation of languages which require unorthodox storage organisations(such as LISP or SNOBOL) is often awkward on conventional machines. The natural store organisation of such languages must be forced onto the rigid underlying store structure. This naturally results in losses, both in store utilisation and in execution speed.

(iii) Machine instruction sets are usually designed as 'general purpose' instruction sets. This usually means that they are not particularly suited to the support of any high-level language. The machine support requirements of different programming systems differ radically and it is impossible to accomodate all of them using a general purpose instruction set.

The B1700 has a bit addressable store where cell sizes may be defined by the s-machine implementor. S-machine instruction sets are implemented via microprograms held in an overlayable control store. Separate instruction sets may be defined to support each programming system on the machine and these may execute concurrently by multiprogramming microprograms in control store.

The freedom made available by the variable memory cell size and definable instruction set offers economic advantages in:-

(i) Store Utilisation

(ii) Program Execution Speed

(iii) System Performance Evaluation

Wilner suggests that the extra computation required to accomodate this flexibility is more than compensated by the above advantages. Each is considered in more detail below.

2.2.1 Store Utilisation

The problem of minimising memory usage is one which has existed throughout the history of computing. Whilst the introduction of virtual memory systems has largely solved the problem as far as the user is concerned, the use of memory must now be managed by the system. Optimum utilisation requires program working sets to be minimised. By physically reducing the store requirements of a program,more usable information can be stored per memory page. Hence the number of pages in the program working set may be reduced.

The representation of information in systems with a fixed memory cell size has a high degree of redundancy. Program data must be fitted to the machine cell size and situations where truth values are stored in 8 bits and small integers in 24 bits are not uncommon. Such situations need not arise on the B1700 as data may be stored in the appropriate number of bits required to hold that information. For example, truth values would be represented by a single bit. As a result, the number of bits required to encode a given amount of data can be significantly reduced in a bit addressable machine.

Compaction may also be achieved in the storage of s-machine code by utilising frequency encoding techniques with instruction op codes. These techniques, suggested by Huffman(H2) involve

There are two primary contributions to this increase in speed:-

(i) Language oriented abstract machine instructions can have a higher semantic content than general purpose machine instructions. This means that a single language oriented instruction may replace a number of general purpose machine instructions.

(ii) As language oriented machine instructions reflect high-level language operations, there are fewer instructions to execute when compared with the same high-level program compiled into a general purpose instruction set.

Implementing language oriented abstract machine interpreters via a microprogram removes the traditional disadvantage of interpretative implementation, that is, the significant decrease in execution speed in an interpreted system. Wilner estimates that microprogrammed interpreters are about 10 times faster than the same interpreters coded in a high-level language.

When a suite of programs was executed on a B1700 and on an IBM S/3 computer of similiar power, the B1700 executed the programs in about half the time taken by the S/3 machine. This illustrates the gain in speed possible with a language oriented machine.

2.2.3 System Performance Analysis

The need for an analysis of program performance is becoming more important as high-level languages displace machine codes for general programming. Knuth(K1) has shown that most programs spend about 90% of their execution time in relatively short code sections.

For optimum performance it is important to identify and 'tune' these sections of code.

Up till now, system performance analysis has been mostly intuitive rather than based on quantitative measurements. This is partially due to the fact that performance analysis can perturb a system to such an extent that the validity of measurements is in doubt. Even if this is not the case, performance measurements usually result in significant systems overhead.

A solution to this measurement problem is to build the measurement tools into the machine hardware, thus avoiding any perturbation of the software performance. This is the approach used on the B1700.

Special purpose monitoring instructions, suitable for gathering information on the performance of executing programs can be included in each s-machine. Wilner estimates that this adds no more than ½% to program execution time. It is likely that this slight overhead can be compensated for by software tuning using the results of the system monitoring.

2.2.4 The B1700 - A Summary

The design of the B1700 computer is a significant advance over most current machine architectures. The machine is designed solely for the implementation of soft machine interpreters and exhibits more flexibility than is usual in computer architectures.

This flexibility is attained by:-

(i) Microprogramming

The B1700 has no fixed instruction set and different machine instruction sets may be implemented for each s-machine.

(ii) Bit Addressable Store

An arbitrary store cell size is not imposed on the s-machine programmer. He may design his s-machine to use the most suitable store cell size.

As a result of this flexibility, more efficient use may be made of the machine store, program execution times may be reduced, and system measurement facilities may be included in each s-machine.

2.3 Microprogramming the B1700

This section surveys microprogramming languages which have been developed or proposed for the B1700 computer series. Three languages, each of which may be regarded as a high-level assembler language, are discussed. These languages are:-

(i) MIL

Burroughs proprietary language developed for microprogramming the B1700.

(ii) BML

A register transfer language whose syntax is based on ALGOL.

(iii) MPL1700

A higher-level language than either BML or MIL but still oriented towards the B1700.

The features of each of these languages will be discussed in turn, with the section concluding with a comparison of MIL, BML, and MPL1700.

Before describing these languages however, we provide a brief exposition of the B1700 micro-architecture. This should enable a reader without detailed machine knowledge to understand examples used in the text. A fuller description of the micro-architecture can be found in Appendix 2 and the definitive description in the B1700 Systems Reference Manual(B2).

2.3.1 The B1700 Micro-architecture

The B1700 has a fairly complex micro-architecture. The machine has a total of 59 registers accessible to an executing microprogram, a 32 element address stack, and a scratchpad of 16 48-bit registers. The scratchpad may also be considered as 32 24-bit registers. Of the 59 registers, there are 4 general purpose registers called X, Y, T, and L. These are 24-bit registers. The remaining registers are dedicated to special purposes. These are:-

(i) Store Addressing

FA and FB, which may be concatenated to form the F register.

(ii) Condition Registers

XYCN, XYST, FLCN, BICN, and INCN. These hold information about the relative states of the X, Y, and F registers and also may act as interrupt registers.

(iii) Result Registers

These hold the results of functions of X and Y computed by a 24-bit function box.

(iv) Microinstruction Addressing

Registers A, M, and MBR.

(v) Control Register

Register C holds information about the kind of data(binary, decimal, etc) currently in use.

(vi) Base and Limit Registers

Registers BR and LR.

The address stack is used when calling micro-routines,with the top of this stack held in a register called TAS. The scratchpad is used for the storage of temporary information and for implementing s-machine registers.

The B1700 is a vertically microprogrammed machine. This means that each microinstruction specifies a single machine operation. Horizontally microprogrammed machines, like some models of the S/360 series, have wider microinstructions in which several operations to be executed in parallel may be specified.

Each B1700 microinstruction is 16 bits wide and there are 36 instructions in the microinstruction set. As the majority of computations involve either the general purpose registers or the F register, there are a number of microinstructions for operating on these registers. For example:-

(i) Shift/Rotate Instructions

Either the X, Y, or T registers may be shifted or rotated as can the concatenated register XY.

(ii) Memory Transfer Instructions

These cause data to be moved to and from memory using the general purpose registers. The memory address is specified in FA and the number of bits to be transferred in FB.

(iii) F Register Instructions

These instructions change the memory address and/or the field length.

In addition, microinstructions exist to transfer data between registers and between registers and the scratchpad. There are instructions for conditional and unconditional branching, call instructions and a special instruction permitting control store to be reloaded dynamically.

2.3.2 MIL

The language MIL(Micro Implementation Language) is a language, developed by Burroughs, to microprogram the B1700. It is fundamentally an assembly language with the majority of MIL statements translated into a single microinstruction. However, there are a number of higher-level facilities in MIL:-

(i) Macros

The user may define simple string replacements or more complex macros which may have parameters.

(ii) Conditional Statements

If-then-else conditional statements are a feature of MIL.

(iii) Block Structure

A limited form of block structure is allowed which defines compound statements and the scope of macro names.

The syntax of MIL is based on English rather than algebraic notation. It has some affinity with COBOL as each MIL statement starts with a reserved verb and 'noise' words are permitted within statements. The language is illustrated by example below, and is fully defined in the appropriate Burrough's reference manual(B3).

Some examples of MIL statements are:-

(i) DEFINE BASE_REG = S1A

This associates the name BASE_REG with the scratchpad location S1A. Subsequent references to BASE_REG will cause it to be replaced by the string "S1A".

(ii) MACRO ADD(A, B, C) =

```
        MOVE A TO X
        MOVE B TO Y
        MOVE SUM TO C$
```

This defines a macro called ADD with formal parameters A, B, and C. The actual parameters would be registers or scratchpad locations.

(iii) READ 8 BITS TO X INC FA

This statement transfers data from memory to the X register. Eight bits are transferred and the memory address register FA is incremented by 8.

(iv)
```
IF XYCN(3) THEN
   BEGIN
       SHIFT X LEFT BY 3 BITS
       COUNT FA DOWN BY 8
   END ELSE
       CALL JUMPOVER
```

This example illustrates the IF statemnt, and the SHIFT, COUNT, and CALL statements. Bit 3 of register XYCN is tested. If it is on, the X register is shifted left and

register FA is decremented by 8. If the tested bit is off, a call is made to the micro-routine JUMPOVER.

2.3.3 BML

BML is a register transfer language for the B1700, designed by De Witt et al(D2). The motivation for its implementation was that, when introduced, MIL was 'company confidential' and MIL programs could not be published.

The language is slightly lower level than MIL, but its syntax is more consistent as it is based on an algebraic notation. To illustrate BML, the examples below display BML commands along with their MIL equivalents.

(i) MIL

```
READ 8 BITS TO X INC FA
```

BML

```
X := MEM[8,FA+]
```

Memory references in BML are made via the reserved word MEM. The bracketed symbols include the parameters of the instruction.

(ii) MIL

```
IF X > Y THEN
   CALL XGREATER
ELSE BEGIN
   MOVE Y TO X
   LIT 0 TO Y
   SHIFT T RIGHT BY 5 BITS
END
```

BML

```
IF X > Y GO TO LAB
   CALL XGREATER
   GO TO LAB1
LAB: X := Y
     Y := 0
     T := SHL T(5)
LAB1:
```

Notice that BML does not have a compound statement facility or an if-then-else statement. This necessitates the use of labels and go to statements to implement simple conditionals.

BML has no macro facilities and lacks MIL's limited block structure. Apart from a slightly neater syntax(at least for non-COBOL programmers) it appears to offer no advantages over MIL.

2.3.4 MPL1700

MPL1700 is a B1700 microprogramming language designed as part of the microprogramming research project at St Andrews University.The language is described by Fisher et al(F1).

MPL1700 is a machine dependent language but at a higher level than both MIL and BML. The language has high-level control statements such as repeat-until, if-then-else, case-of, etc. There is a simple macro facility, variables may be declared and structure may be ascribed to these variables.

An example of this latter facility is:-

structure listelem = (car(16),cdr(16))

This indicates that a list element is a 32-bit quantity. The first 16 bits are referred to as 'car' and the second 16 'cdr'.

Similiarly arrays of structures may be defined:-

array 26 structure list = (car(16),cdr(16))

This defines an array of 26 structures which may subsequently be associated with some name.

Structures are associated with names as follows:-

list baseregs

This specifies that the variable baseregs should be considered of type list.

Individual elements of the structure may be accessed via the associated name:-

baseregs(7).cdr

This accesses the second 16 bits of the 8th element of baseregs.

MPL1700 statements are register transfer statements allowing both right and left assignment. Some examples are given below:-

(i) T := S1A => LR

This specifies that the contents of S1A be moved to T and also to LR

(ii) if L < 10 then

{ if T = 4 then T := 0 else T := 1 }

An MPL1700 conditional statement whose meaning should be fairly evident.

(iii) while T < 10 do

T := T + a + 5

Similiarly, an MPL1700 loop with a fairly obvious meaning.

MPL1700 is an interesting attempt to raise the level of microprogramming languages although it is difficult to envisage the language implementation producing microcode as efficient as that possible using a lower-level language. This, however, can only be demonstrated when the language implementation is complete.

2.4 Microprogramming Languages for Other Machines

In this section, we examine a number of other microprogramming languages which have been developed to microprogram various computers.

Recent trends in microprogramming language design have been away from the complex, difficult to read, micro-assemblers towards higher-level, more readable languages. Generally, however, commercially developed languages are still fairly low-level - the level being approximately that of MIL.

In a university environment, some research work has been carried out in developing a high-level microprogramming language for an Interdata 3 computer. This language, based on PL/1 is described in section 2.4.3 below.

Firstly, we examine two microprogramming languages which illustrate the trend towards higher-level microprogramming. These languages are:-

(i) The Datasaab FCPU microprogramming language.

(ii) The MLP-900 microprogramming language.

2.4.1 The Datasaab FCPU Microprogramming Language

The microprogramming language for the Datasaab FCPU(Flexible Central Processing Unit) has been described by Lawson and Blomberg(L3). It was designed to facilitate writing and reading microprograms, as it was anticipated that more microcode would

be written for the FCPU than for previous medium-scale machines.

The language has simple block structures allowing local and global declarations, do-loops, and a case statement. In spite of this, the language is still a low-level language as the FCPU machine registers may be directly accessed. The example below provides an illustration of the form of the language.

```
/*      Emulation of Datasaab D23 processor          */

D23OP:     DO  CASE (8)
OP00:      DO
           .
           .
           .
           END

           .
           .
           .

OP07:      DO                  /*    Add instruction   */
              AR = ADD(AR,AUTFR,0)
              SS-S-IND = AU-OVERFLOW
              FUTFR = AR
              WRITE(ACR.1,FUTFR.0,LEFT) LENGTH (24)
              VLS-I-IND = AU-SIGN
              START(TARGET,D23OP)
           END
           END
```

An Example of the Datasaab FCPU Microprogramming Language

The case switch is made on the basis of a register value with the number of possible cases indicated by the bracketed figure following the reserved word CASE.

According to Lawson, the use of this language has improved initial program construction and subsequent debugging, as well as simplifying microprogram maintenance.

2.4.2 The MLP-900 Microprogramming Language

The MLP-900 was a fairly early(1970) user-microprogrammable machine. The machine design is described by Lawson and Smith(L4) and its microprogramming language by Oestricher(O2).

This microprogramming language, called GPM, is a register transfer language but it has high-level control statements and a means of writing simple arithmetic expressions.Some examples of GPM statements are given below:-

```
F1 ← ( F1 AND F2 ) OR ( F3 AND F4 )
```

This is a register assignment statement assigning the value of the logical expression to register F1.

```
DO
  F0 ← F1 + F2
  IF F0 > F4 BREAK
END
```

This is a GPM loop. Notice that the DO-END pair delimit an unbounded loop with loop exit made via the BREAK statement.

Other control statements in GPM include an if-then-else statement and a case statement. Unfortunately, Oestreicher's paper on GPM is rather brief and it has not been possible to obtain fuller information about the language.

2.4.3 MPL - A Machine Independent Microprogramming Language

MPL is a 'machine independent' microprogramming language developed by Eckhouse(E1) for the Interdata 3 computer. Its syntax is based on PL/1. Eckhouse has also surveyed a number of other microprogrammable machines and maintains that reasonably efficient implementations of MPL could be devised for them.

MPL has facilities for declaring data area names and for defining the width of these data areas. There is no mechanism for ascribing structure to the data areas. The language facilities include procedures, expressions, and control statements such as if-then-else, do-while, and go to. Some examples of MDL statements and declarations are:-

(i) DCL (R0,R1,R2) BIT(12)

This declares three 12-bit registers R1,R2, and R3.

(ii) R0//R1 = R0//R1 + 2

This specifies that registers R0 and R1 are to be concatenated and that the contents of the concatenated register are to be increased by 2. The concatenation operator is //.

(iii) IF CARRY THEN GO TO RXFORM

This tests the logical variable CARRY. If true, control is transferred to the label RXFORM. CARRY is declared as an EVENT type which may be mapped onto the particular bit in the underlying machine which detects carries.

Eckhouse has implemented MPL using a three phase system. Firstly, the language is translated into a machine independent intermediate code. Eckhouse claims that this code may be mapped onto a number of different microprogrammable machines. This intermediate code is translated by phases 2 and 3 of the compiler into Interdata 3 microcode.

The system appears to be successful but, as the author admits, the Interdata machine has a very simple microinstruction set and micro-architecture. As MPL was designed to produce emulators for 'hard' machines no information on its suitability for the building of language oriented soft machines is available. Unfortunately there appears to have been no further work done with MPL since Eckhouse's thesis and, as a result, language evaluation is a difficult task.

2.5 Summary

This chapter has covered background material which is relevant to our project. Much of the research in this field has taken place on an 'ad hoc' basis. With the exception of Wortman's MDL and, to a lesser extent, Eckhouse's MPL little direct contribution was made to our work by previous research. Four distinct topics have been covered:-

(i) Machine Description Languages

(ii) The Design of the B1700 Computer

(iii) Microprogramming the B1700

(iv) Other Microprogramming Languages

Section (i) reviewed a number of machine description languages. Most of these are designed as 'hard' machine description languages and are not particularly suitable for the description of s-machines. One s-machine description language, based on PL/1, is described.

Section (ii) examined the design of the B1700 and how that machine is designed to support s-machine interpreters. The B1700 has a store which is bit adressable and may be user microprogrammed. The microinstruction set has been designed as a general purpose instruction set for constructing a variety of s-machine interpreters. The flexibility offered by these features has advantages in memory utilisation and increased program execution speed.

The remainder of the chapter is devoted to a study of microprogramming languages. Firstly, three microprogramming languages for the B1700 are described. These are all machine dependent languages with some high-level language features. Of these languages, MIL and BML are basically assembly codes but MPL1700 adopts a somewhat higher approach to microprogramming the B1700.

Finally, we examined microprogramming languages available for other machines. These exhibited a definite trend towards a higher level and the final microprogramming language studied, MPL, appears to be a true high-level language. Unfortunately,development of MPL does not appear to have continued beyond the stage of investigating its feasability.

From our survey of background work on machine description and microprogramming languages a number of conclusions can be drawn. These are:-

(i) The trend in microprogramming appears to be towards higher-level languages which are machine dependent. PL/1 appears to be the dominant language influencing the syntax of these microprogramming languages.

(ii) Machine description languages have been designed primarily by digital engineers for describing current hardware. With the exception of Wortman's MDL(again PL/1 based) these machine description languages are not particularly suitable for the description of more complex, language oriented soft machines.

(iii) Apart from Eckhouse's pioneering MPL there seems to have been no attempts made to develop a true high-level microprogramming language. Indeed, a recent conference discussion, chaired by Lawson(L5) regarded the prospect with horror because of possible inefficiencies.

Our survey of background material convinced us that investigating the possibilities of a combined microprogramming and s-machine description language was a useful aim.

Note added in preparation

A very recent publication by Dewitt(D6) describes a proposed high-level microprogramming language which is, in many respects, similiar to ours. Special attention has been taken over the problem of generating efficient microcode, but the language implementation is not yet complete.

CHAPTER 3

SIMULATING THE B1700 SYSTEM

This chapter is a description and evaluation of a simulation package for the B1700 micro-computer system. This simulation system is comprised of two programs:-

(i) A hardware simulator.

(ii) A compiler for MIL, Burroughs microprogramming language.

The simulation system is described in a number of sections:-

(i) A simulator description. This presents an overview of the hardware simulator and describes the simulation of the B1700 arithmetic unit and memory management system.

(ii) Simulator extensions.This describes a number of facilities which we have added to the simulator for simplifying the collection of diagnostic information.

(iii) The MIL compiler. This section is an overview of the system used to compile MIL and a description of an intermediate machine code used in the compilation.

(iv) A review and evaluation of the system. After presenting examples of system output, the performance of both the simulator and the MIL compiler is described. Defects and possible system improvements are discussed.

However, before covering this material, it is apposite to examine the reasons why this part of the project was undertaken.

As explained in Chapter 1, the work described here is part of a larger research project investigating the construction of s-machines. This larger project was initially based on the B1700 computer. The author of this thesis was involved at an early stage of the project before any hardware became available. The decision to build a B1700 simulator was taken at this stage, for two reasons:-

(i) It would provide an opportunity for the author and others working on the project to familiarise themselves with the B1700 micro-hardware. This was important in view of the projected development of our research towards a high-level microprogramming language.

(ii) A simulation system would allow microprograms to be constructed, tested, and evaluated without recourse to (unavailable) B1700 hardware. The debugging and evaluation advantages of a simulation system compared to a hardware system are well known.

Accordingly, the system described below was built and, in general, has satisfied our initial aims. The implementation of the simulation system unquestionably provided in-depth knowledge of the machine hardware which has been essential for later parts of the project.

The system has been used to develop and test microprograms but, unfortunately, system considerations dictate that it executes under a batch operating system. As interaction with the system is impossible, its use as a debugging tool is necessarily limited.

3.1 An Overview of the Hardware Simulator Program

In this section, we present an overview of the hardware simulation system. The general structure of the program is described along with the simulation of memory addressing and the arithmetic and logical function box. For a fuller description of the simulator, the reader is referred to the document by Sommerville(S1) which provides a complete description. The B1700 micro-architecture is described in Appendix 2 and, more fully, in the B1700 System Reference Manual(B2).

3.1.1 The Structure of the Simulator Program

The B1700 simulator is implemented in ALGOLW and runs on an IBM S/360 computer. The program interprets B1700 microinstructions, and, therefore, has the usual interpreter structure. That is, the main program consists of a loop, fetching and decoding each microinstruction and then switching to the appropriate code to interpret that instruction.

The main simulator algorithm is shown below:-

```
while simulating do
    {
        fetch and decode instruction
        case
            op code = 1: REGISTER MOVE
            op code = 2: SCRATCHPAD MOVE
            .
            .
            op code = 0004: NORMALISE
        endcase
            else error
    }
```

The Structure of the B1700 Simulator

Programming the simulator presented few problems as each microinstruction has a clearly defined function with no intra-instruction parallelism. However, the simulation of the arithmetic unit and its associated parallelism is of interest as is the simulation of the B1700 memory addressing scheme. These are described in more detail below.

3.1.2 Simulating the Arithmetic and Logical Unit

The arithmetic and logic unit of the B1700 is a hardware module which accepts as input the general purpose registers X and Y, along with the control register C. The output from this unit consists of various arithmetic and logical functions of X and Y, such as sum, difference, logical and, etc. These functions are automatically computed in parallel whenever the contents of X, Y, or C is changed. The C register provides information about the type and length of the operands in X and Y.

Obviously, in a serial program such as the B1700 simulator, it is impossible to simulate factually the parallel operation of this unit. A different approach, producing the same result, was therefore adopted.

It is clear that the recomputation of all arithmetic and logical functions whenever X, Y, or C is changed is unnecessary. The X and Y registers may be used for purposes not requiring ALU results and, when such results are required, generally only one function is needed. This fact is used in the technique adopted to simulate the ALU.

When a microinstruction calls for an ALU function result(these are held in special purpose registers), this call is detected and an ALGOLW function is called. This function computes the appropriate arithmetic or logical function required using the X, Y, and C registers. Thus the operation of the ALU may be simulated without parallelism and without unnecessary computation being carried out.

3.1.3 Simulating the B1700 Memory Management System

In a B1700 processor, all memory operations are channelled through a hardware module known as the memory control unit(MCU). The function of this unit is to provide memory bit addressability for the executing microprogram. Memory is, in fact, organised in bytes and the MCU must resolve the difference between the bit address used by the microprogram and the actual byte address.

Main store on the B1700 is addressed by a 30-bit address, divided into three fields.

(i) Bits 0-23

The absolute bit address in memory

(ii) Bits 24-28

The number of bits required(0-24)

(iii) Bit 29

A direction bit indicating whether the address refers to the top or bottom bit of the element being fetched.

The MCU fetches the byte addressed by the most significant 21 address bits plus three bytes above or below it in memory, depending on the field direction bit. The appropriate number of bits are then selected from this group of four bytes.
The simulation of the MCU is carried out by an ALGOLW procedure called MEMORY_CONTROL.

MEMORY_CONTROL takes three parameters:-

(i) The general purpose register used as a source or destination register for the data transferred from memory.

(ii) The number of bits to be fetched from memory.

(iii) The operation to be executed(read, write, or swap).

Notice that a field direction is not specified. This is unnecessary as the address register FA is modified at an earlier stage in the simulation if the field direction is negative. Register FA, therefore, always refers to the most significant bit of the element in store. The algorithm describing the procedure MEMORY_CONTROL is shown below:-

```
MEMORY_CONTROL

        %   Parameters are: a general purpose register
        %                 : the number of bits to be fetched from store
        %                 : the operation to be executed
        %   Local variables: MIR, MIR2

        Compute byte address from FA
        MIR := Fetch 4 bytes from store
        Compute bit address using least significant 3-bits in FA
        if  operation = read  then
           {
               Transfer appropriate number of bits from MIR to
               specified general purpose register
           }
```

```
else
    if operation = write then
        {
            Enter appropriate number of bits into MIR from
            general purpose register
            Rewrite MIR to memory
        }
    else
        {
            %   Swap operation
            MIR2 := MIR
            Perform write operation
            MIR := MIR2
            Transfer appropriate number of bits to general
            purpose register
        }
END MEMORY_CONTROL
```

The Algorithm used in the Simulation of the B1700 Memory Control Unit

3.2 Extensions made to the B1700 Simulator

A simulator program designed for debugging and program evaluation should contain facilities over and above those provided by the bare machine hardware. These facilities should allow diagnostic information to be collected and should gather information on the performance of the machine. Accordingly, additional microinstructions have been added to the B1700 simulator instruction repertoire and system measurement routines have been designed for the simulator. These are described below.

3.2.1 Additional Simulator Microinstructions

Microinstructions have been added to the simulator to perform the following functions:-

(i) To dump the contents of specified storage areas to the line printer

(ii) To provide simple read and write input/output operations

These added microinstructions permit the programmer to display the contents of B1700 storage areas during the testing and debugging phase of microprogram development.

Dump Microinstructions

These instructions may be included anywhere in a microprogram and allow the following operations to be carried out:-

(i) Dump control memory

(ii) Dump main memory between the base and limit registers

(iii) Dump all registers, the scratchpad, and the stack

(iv) Dump the machine address stack

(v) Dump the scratchpad

(vi) Dump the general purpose and address registers

The provision of these operations considerably simplified the testing of the simulator and the development of microprograms. Until the I/O operations described below were implemented the dump instructions were the only means of output from the machine.

Input/Output Microinstructions

Input/output on the B1700 is normally descriptor based and requires a considerable amount of operating system intervention. It was not thought appropriate to include such a general purpose system in our simulator.

Thus, a very simple I/O scheme was devised primarily to simplify microprogram testing. Two additional instructions were added to the microinstruction set.

These instructions are:-

(i) A read instruction

(ii) A write instruction

When using these instructions, the microprogrammer specifies the number of bytes to be input or output, the store address of the information to be transferred and a peripheral device number. This information must be loaded into general purpose machine registers. The read and write microinstructions extract this information from the appropriate registers and carry out the data transfer.

3.2.2 System Measurement

The simple provision of facilities for system measurement and performance evaluation is a significant advantage offered by a simulation system compared to a 'hard' system. Facilities of this type which have been designed for inclusion in the B1700 simulator are:-

(i) A means of counting the number of accesses to each machine register.

(ii) A means of counting the number of accesses made at a particular store address.

(iii) A count of the number of times each instruction in the microinstruction set is used.

To date, only the latter facility has been implemented. With the information obtained from the microinstruction count the simulator calculates the total time(in B1700 clock cycles) taken for program execution.

The implementation of the location access count features in the simulator is a straightforward process. A simulator directive

COUNT <parameter list>

specifying the locations and registers to be monitored could be introduced. This facility would enable the user to optimise his use of registers and store.

3.3 A Translator for MIL

This section of the thesis discusses the reasons for implementing an MIL compiler, and gives a brief overview of the structure of that program. For a fuller description of the techniques used in compiling MIL, the reader is referred to Sommerville(S2).

The language MIL is a low-level microprogramming language which has been used to implement all the manufacturer supplied microprograms implemented on the B1700. The language is briefly described in chapter 2 of this thesis and definitively in the appropriate Burroughs reference manual(B3). Fundamentally, it is an assembler language with a COBOL-like syntactic structure. That is, an English-like rather than an algebraic syntax notation is used. MIL statements generally have a one-to-one correspondence with B1700 microinstructions.

However, the language has a number of high-level features:-

(i) Programmer defined macros

(ii) An if-then-else conditional statement

(iii) A limited form of block structure where macro names may be localised.

The implementation of a compiler for MIL is therefore more complicated than the construction of a simple assembler. However, the work involved is in no way comparable with implementing a compiler for a high-level language.

The decision to undertake the implementation of an MIL compiler was made for the following reasons:-

(i) The B1700 simulator was in the final stages of implementation and a means of writing test microprograms was required. Previously, a very simple assembler had been used to write such test programs but using this was a tedious and error-prone procedure. This assembler language was later modified to become IML, the code generated by the MIL compiler.

(ii) As part of our larger research project, it was decided to implement various s-machines as B1700 microprograms. A language was required for writing these microprograms.

(iii) The construction of an MIL compiler would provide experience in using various translation techniques. It was envisaged that this would be useful at a later stage of this project.

3.3.1 The Translation Program

The system which was implemented to compile MIL to microcode is a two-pass system. The first pass accepts MIL statements as input and generates an intermediate code called IML. This is translated to binary microcode by the second pass.

There were two reasons for adopting this two-pass organisation:-

(i) The syntax of MIL is such that SNOBOL4, with its powerful pattern matching facilities, is a very suitable language to use in the translation of MIL. However, it is not particularly suitable for the generation of binary code strings.

(ii) A translator for IML could be easily produced by modifying an already existing program. The construction of the second pass of the MIL translation system was therefore a simple operation.

SNOBOL4 is a very suitable language for processing MIL statements because of its ability to handle MIL's English-like syntax. Most MIL statements contain 'noise' words such as BY, TO, FROM, etc whose function is merely to act as separators. SNOBOL's pattern matching features simplify the recognition and removal of such words.

A feature of MIL is the alternative notations allowed for various constructs, especially the conditional statement. For example, a

simple test for equality of machine registers X and Y may be expressed in four different ways:-

(i) IF X = Y THEN

(ii) IF X EQL Y THEN

(iii) IF XYCN(2) THEN

(iv) IF XYCN(2) TRUE THEN

Naturally, such a plethora of notations causes compilation problems. However, SNOBOL's pattern matching features may be used to convert all alternative notations to that shown in (iv) above before any syntactic or semantic analysis. Notation (iv) is the most general form of the conditional statement.

The removal of 'noise' words and the simplification of alternative notations is carried out by the scanning procedure in the MIL compiler. This procedure also deals with the expansion of macros using the well known input stack technique.

After this preprocessing, the translation of MIL statements is essentially the same as that carried out in a conventional assembler. The first word of each statement uniquely identifies the microinstruction to be generated.

The basic translation algorithm is shown below:-

```
COMPILE_MIL

        initialise compiler
        while  compiling  do
          {
              Abstract first word of MIL statement
              case
                  word = "MOVE" : REGISTER MOVE INSTRUCTION
                  word = "LIT"  : LITERAL MOVE INSTRUCTION
                  .
                  .
                  .
                  word = "SHIFT" : SHIFT OPERATION
              endcase
                  else error
          }
END COMPILE_MIL
```

The Basic Translation Algorithm used in the MIL Compiler

Having identified the statement to be translated, it is checked for semantic and syntactic errors. If it is free of errors, the statement is converted to its IML form. This conversion is straightforward as IML expects register names, literals, etc to be used rather than some more concise notation. The intermediate language IML is described briefly below.

3.3.2 Intermediate Machine Language

Intermediate Machine Language is a very simple assembly language for the B1700 system. It was originally devised to ease the testing of the B1700 hardware simulator, but is now used as the code generated by both the the MIL compiler and the SUILVEN compiler(described in Chapter 5).

There is a one-to-one correspondance between binary microcode and IML instructions. Each IML instruction has the form:-

<op code>[,<register list>][,<variant list>]

where enclosure in square brackets indicates that the enclosed elements are optional depending on the individual instruction. The op code is a numeric(not mnemonic) code defining the instruction, with the register list being a list of register names. The variant list is a list of numbers representing possible variants for each instruction. IML has no labels - all jumps are encoded as numeric displacements.

To illustrate the language, some examples of IML instructions along with their MIL equivalents are shown below.

	IML	MIL
(i)	1,X,Y	MOVE X TO Y
(ii)	12,15	GOTO LABEL
(iii)	8,FL,H18	MOVE HEX 18 TO FL

3.3.2 Intermediate Machine Language

Intermediate Machine Language is a very simple assembly language for the B1700 system. It was originally devised to ease the testing of the B1700 hardware simulator, but is now used as the code generated by both the the MIL compiler and the SUILVEN compiler(described in Chapter 5).

There is a one-to-one correspondance between binary microcode and IML instructions. Each IML instruction has the form:-

<op code>[,<register list>][,<variant list>]

where enclosure in square brackets indicates that the enclosed elements are optional depending on the individual instruction. The op code is a numeric(not mnemonic) code defining the instruction, with the register list being a list of register names. The variant list is a list of numbers representing possible variants for each instruction. IML has no labels - all jumps are encoded as numeric displacements.

To illustrate the language, some examples of IML instructions along with their MIL equivalents are shown below.

	IML	MIL
(i)	1,X,Y	MOVE X TO Y
(ii)	12,15	GOTO LABEL
(iii)	8,FL,H18	MOVE HEX 18 TO FL

3.4 The Output Produced by the Simulation System

In this section, an example of the output produced when a microprogram is compiled and executed on our system is presented. The example consists of a micro-routine to convert numbers input from EBCDIC representation to their appropriate numeric equivalent. This is a commonly required operation in any high-level language. In the example below, appropriately placed DUMP microinstructions show the changing values of the simulator registers.

```
1           *               DEFINE SOME MACROS TO ILLUSTRATE DEFINE STATEMENT
2
3                           DEFINE X.LT.Y=XYCN(1) TRUE#
4                           DEFINE X.EQ.Y=XYCN(2) TRUE#
5                           DEFINE POPSTACK=MOVE TAS TO#
6                           DEFINE BLANK=H40#
7
8           *               DUMP THE CONTROL STORE AND SKIP OVER MICRO ROUTINES
9
10                          DUMP CONTROL
11                          GOTO START
12
13          *               ROUTINE TO MULTIPLY THE X REGISTER BY 10 LEAVING
14          *               THE RESULT IN X
15          MULT10          MOVE X TO Y                         *  COPY X
16                          SHIFT X LEFT BY 3 BITS              *  MULTIPLY X BY 8
17                          MOVE SUM TO X                       * ADD ORIGINAL X TWICE
18                          MOVE SUM TO X
19                          EXIT                                * FROM MULT10
20
21          *               GETINTEGER ROUTINE TO CONVERT FROM EBCDIC TO BINARY
22
23          GETINTEGER      MOVE B00011000 TO CP                * BINARY OPERANDS
24                          SHIFT Y LEFT BY 3 BITS              * CONVERT BYTE ADDRESS
25                          MOVE Y TO FA                        * TO BIT ADDRESS
26                          MOVE 0 TO TAS                       * PUT ZERO ON STACK
27          *               TAS IS USED AS AN INTERMEDIATE REGISTER
28          READCHAR        READ 8 BITS TO X INC FA             READ A CHARACTER
29                          MOVE BLANK TO Y                     * IF IT'S A BLANK
30                          IF X.EQ.Y THEN GOTO BLNKFND         * WE ARE FINISHED
31                          MOVE 239 TO Y                       * CHECK CHARACTER IS A
32                          IF X.LT.Y THEN GOTO NONNUMB         *  NUMBER
33                          LIT 240 TO Y                        * CONVERT EBCDIC TO
34                          MOVE DIFF TO L                      *  DECIMAL-PUT IN L
35                          POPSTACK X                          * GET LAST INTERMEDIATE
36                          CALL MULT10                         * RESULT AND MULTIPLY
37                          MOVE L TO Y                         * ADD DIGIT FOUND AND
38                          MOVE SUM TO TAS                     * MOVE SUM TO TAS
39                          GOTO READCHAR                       * GET NEXT CHARACTER
40          *               NOW AT END OF NUMBER, DUMP MAIN PROCESSOR REGISTERS
41          BLNKFND         DUMP MAINREGS
                                                                * MOVE FINAL RESULT TOX
```

```
43                      EXIT
44          NONNUMB     MOVE 1 TO X                     * PUT ERROR CODE IN X
45                      POPSTACK Y                      * CLEAR STACK
46                      CALL ERROR
47                      EXIT
48
49          *           ERROR ROUTINE IS NOT IMPLEMENTED.IT MERELY MOVES 999 TO Y
50          *           TO INDICATE AN ERROR HAS OCCURRED AND DUMPS THE REGISTERS
51
52          ERROR       MOVE 999 TO Y
53                      DUMP REGISTERS
54                      EXIT
55
56          *           THIS IS THE START OF THE PROGRAM
57
58          START       LIT 0 TO X                       * SET UP I/O INSTR.
59                      LIT 5 TO L                      * X HAS ADDRESS,L THE
60                      CRDIN                           * NO OF CHARS READ
61                      CLEAR Y                         * SET Y TO ZERO
62                      CALL GETINTEGER                 * CONVERT TO BINARY
63                      DUMP MAINREGS
```

CONTROL STORE DUMP

0	#00000005	#0000C022	#000010A1	#00000403	#000010E0	#000010E0
6	#00001BA4	#00008C18	#00000423	#000011A8	#00008B00	#00007108
12	#00008140	#00004CC1	#0000C00A	#000081EF	#00004CA1	#0000C00A
18	#000081F0	#000018E3	#00001BA0	#0000F014	#000013A1	#000010EB
24	#0000D00E	#00000008	#00001BA0	#00001BA4	#C0008001	#00001BA1
30	#0000E001	#00001BA4	#00009100	#000003E7	#00000004	#00001BA4
36	#00008000	#00008305	#00000B00	#00000320	#0000F022	#00000008
42	#0000FFFF					

********** DUMP OF MAIN PROCESSOR REGISTERS **********

GENERAL PURPOSE REGISTERS	X =	#00000040	Y =	#00000040
	L =	#00000003	T =	#0003001B
ADDRESS REGISTERS	FA =	#00000020	FB =	#FBFBFBFB
INSTRUCTION REGISTERS	M =	#00000008	A =	#0000001A
BASE/LIMIT REGISTERS	BR =	#00000000	LR =	#00000000
TOP ELEMENT OF ASTACK	TAS =	#00000355		

********** DUMP OF MAIN PROCESSOR REGISTERS **********

GENERAL PURPOSE REGISTERS	X =	#00000355	Y =	#00000040
	L =	#00000003	T =	#00030018
ADDRESS REGISTERS	FA =	#00000020	FB =	#FBF8FBFB
INSTRUCTION REGISTERS	M =	#00000008	A =	#0000002A
BASE/LIMIT REGISTERS	BR =	#00000000	LR =	#00000000
TOP ELEMENT OF ASTACK	TAS =	#00000000		

MICRO-INSTRUCTION

HALT
OVERLAY
NORMALISE
CASS CONTROL
BIAS
STORE F
LOAD F
SET CYF
SWAP MEMORY
CLEAR REGISTERS
SHIFT/ROTATE X OR Y
SHIFT/ROTATE X AND Y
COUNT FA/FL
EXCHANGE DOUBLE WORD
SCRATCHPAD RELATE FA
MONITOR
REGISTER MOVE
SCRATCHPAD MOVE
FOUR BIT MANIPULATE
BIT TEST BRANCH
SKIP WHEN
READ/WRITE
MOVE 8 BIT LITERAL
MOVE 24 BIT LITERAL
SHIFT/ROTATE T REGISTER
EXTRACT FROM T REGISTER
BRANCH
CALL
DUMP REGISTERS
DUMP CONTROL
MULTIPLY
DIVIDE
DUMP MAIN REGISTERS
DUMP STACK
DUMP SCRATCHPAD
COREDUMP
I/O READ
I/O WRITE

NO OF EXECUTIONS

0
0
0
0
0
0
0
0
0
1
4
0
0
0
0
0
27
0
0
7
0
4
14
0
0
0
5
4
0
1
0
0
2
0
0
0
1
0

```
NO OF CLOCK CYCLES          86

***** END OF SIMULATION-- ALL MICRO INSTRUCTIONS PROCESSED *****
```

3.5 An Evaluation of the Simulation System

This section of the system description presents some figures regarding the speeds of the B1700 simulator and the MIL compiler. It also examines defects in the overall system and suggests how these defects might be remedied.

3.5.1 The Performance of the System

The figures presented below have been gathered by executing microprograms of various sizes on the simulation system.

Average number of microinstructions executed per second on the B1700 hardware simulator	1000
Theoretical number of microinstructions executed on a real B1700 processor	6×10^6

The great discrepancy in execution speeds is to be expected for two reasons:-

(i) Interpretative execution of machine instructions is always much slower than hardware execution of the same instructions

(ii) The semiconductor technology used in the construction of the B1726 store is much more modern than that used in the IBM S/360/44. Hence the B1700 is inherently a faster machine than the 360. Therefore, even if microinstructions were executed at hardware speed, the discepancy between the B1700 and the 360 would still exist.

The MIL compiler has been evaluated in a similiar fashion by compiling a number of microprograms. These were also compiled using Burrough's MIL compiler on a B1726 with 48k bytes of store.

Average number of MIL statements translated per minute by the S/360 version of the MIL compiler	160
Average number of MIL statements translated per minute by the B1726 MIL compiler	50

Notice that, in this case, the combined SNOBOL4/ALGOLW system is about 3 times faster than the MIL compiler on the B1700. This is due to the fact that the comparison was made using a B1700 with only 48k bytes of store. This was the only machine available. 48k bytes is an inadequate amount of store to run the B1700 efficiently and thrashing is a serious problem. In fact, most of the compilation time on the B1700 is spent transferring segments from disk.

3.5.2 Defects and Improvements

Naturally the system as it stands contains a number of defects. However, it must be borne in mind that the simulation system was constructed as part of a research project where it was important to produce a usable system in the shortest possible time.

Of the failings of the hardware simulator, the most serious is that it is merely a hardware simulator. No attempt is made to simulate the actions of the B1700 operating system(MCP).

The MCP interacts to a significant extent with microprograms executing on a B1700. The lack of this interaction is a serious drawback, when the simulator is used to test microprograms which are subsequently to run on a real machine. This was recognised when the simulator was being programmed but no operating system simulation was included for the following reasons:-

(i) The details of the interaction of the MCP and an executing microprogram are shrouded in mystery. Documentation is almost non-existent and to discover the extent of the interaction would have involved much study of the source code listing of the MCP.

(ii) When the simulator was under construction, it appeared that the inclusion of any operating system simulation would significantly degrade the program performance.

In retrospect, it is probably possible to modify the simulator program so that it may mimic the responses of the MCP when a call is made to the operating system by a microprogram. If such a feature were included, simulator tested microprograms would be more compatible with those for a real machine.

Other, lesser, incompatibilities of the simulator are a result of the fact that the simulator must run under a batch operating system. These incompatibilities are:-

(i) Microprogram loading, normally carried out via a cassette, may not be simulated.

(ii) The machine panel display may not be simulated.

(iii) External interrupts to executing s-machines are not possible.

We see no way of rectifying these defects apart from running the simulator on a dedicated machine.

The MIL compiler suffers from surprisingly few incompatibilities and defects considering the time taken to implement the system, roughly two months. The need to implement the system quickly was directly responsible for those drawbacks which exist in the MIL compiler. These drawbacks are:-

(i) The compiler does not optimise the generated microcode.

(ii) The error messages produced by the compiler are not particularly informative and error recovery is poor.

As our system is a research tool, we do not consider time spent rectifying these defects to be justifiable.

3.6 Summary and Conclusions

This chapter has described and evaluated a simulation of the microprogramming environment of the B1700. The system was designed to familiarise the author with this environment and to enable the testing of B1700 microprograms.

Two distinct programs make up the simulation system:-

(i) A hardware simulator. This program is encoded in ALGOLW and interprets B1700 microinstructions. A number of additions have been made to the simulator to aid the debugging and testing of microprograms.

(ii) A compiler for Burrough's microprogramming language MIL. This compiler is encoded as a two-pass system. The first pass converts MIL statements to an intermediate representation with the second pass converting this representation to binary microcode.

It is fair to say that the system, as it stands, has met our expectations and serves a useful purpose as a tool in our research project.

Constructing the system indubitably provided extremely useful background information on the B1700 microprogramming environment. Although it is far from ideal for microprogram testing, the simulator is adequate for this purpose, and for programs encoded in MIL the conversion to a real machine is fairly trivial. This is due to the fact that the B1700 MIL compiler generates the machine interface information and it is transparent to the programmer.

Although the overall system performance could be improved, we do not consider that time spent on this improvement would be justified. As the system is essentially a research tool, a production standard system is unneccessary and we believe that the time necessary to attain such a standard may be better employed on other projects.

CHAPTER 4

THE PROGRAMMING LANGUAGE SUILVEN

The programming language SUILVEN is a high-level language designed for describing and implementing abstract machine interpreters. In this chapter, the design of the language and the factors which influenced that design are described.

A SUILVEN program is compiled into microcode for the B1700 computer. As discussed previously, the purpose of designing and implementing SUILVEN was to investigate the possibilities of high-level microprogramming.

At the outset of the project, we were aware of the proliferation of programming languages currently in use, and hesitated to add yet another language to that ever increasing pool. However, for the reasons below, we found it desirable to design a new language, rather than produce a compiler for an existing language. The reasoning behind this decision is:-

(i) Apart from Wortman's MDL(W2), which is a high-level s-machine description language, machine description languages appear to be designed for 'hard' rather than soft machine description. As a result, they contain features such as timing facilities which are inappropriate in an s-machine description language. In addition, we considered the control constructs in these languages to be inadequate for producing well-structured programs.

(ii) Wortman's MDL, designed solely for describing s-machines is not easily compiled into microcode. The language is informal in nature which is, of course, acceptable in a description language but not suitable for a language which is to be compiled.

It was therefore decided to design a programming language for s-machine description which may be compiled to microcode. The language, loosely based on MDL, was to be such that a compiler could be produced fairly easily and, as a result, language design and implementation proceeded together for a time. However, before describing this language, subsequently named SUILVEN, some general aspects of programming language design are considered.

4.1 Programming Language Design

In designing a high-level programming language there are a number of trade-offs which must be made between efficiency, convenience, and reliability. Up until recently, efficiency was generally regarded as the most important of these factors. However, Dijkstra and others(D3,D5) have suggested that convenience and reliability are now more important than efficiency. This is due to the increasing discrepancy between the cost of software and the cost of hardware. The cost of writing and maintaining a programming system is now much greater than the cost of the hardware running that system and any programming language features or programming techniques which decrease software costs are desirable.

General programming language features which are concerned with efficiency, convenience and reliability are:-

(i) Storage Allocation Strategies

(ii) Language Control Constructs

These topics are discussed below.

4.1.1 Storage Allocation

High-level language storage allocation schemes fall into two categories:-

(i) Compile-time(static) storage allocation

(ii) Run-time(dynamic) storage allocation

The former scheme allocates absolute addresses to variables at compile time, whilst the latter allocates relative addresses at compile-time. These are converted at run-time to absolute machine addresses.

Dynamic storage allocation is the method adopted by programming languages such as ALGOL60. It has several advantages over a static allocation scheme, viz:-

(i) Recursion may be implemented in a clean and straightforward manner.

(ii) Dynamic arrays may be included as a feature of the programming language.

(iii) Storage may be shared as local variable storage may be allocated and de-allocated.

However, the penalty for these advantages is a reduction in program execution speed, as run-time intervention by the programming system is needed to handle storage allocation. This penalty is avoided with a static allocation system, although at a cost of a more inflexible system for the user.

4.1.2 Program Control Constructs

The control constructs available in a programming language significantly affect the convenience of programming in that language and the readability of programs written in that language. If the flow of control during program execution matches the static representation of the program, understanding and debugging the program is considerably simplified.

The control statement GO TO was first condemned by Dijkstra(D3) and shown to be unnecessary by Bohm and Jacopini(B4). The statement has been condemned because of its power to transfer control to anywhere within a program. The execution sequence of a program with GO TO statements is not necessarily "top-to-bottom" and, as a consequence, predicting the behaviour of such a program is often difficult. As well as this, the inclusion of GO TO statements render ineffective recently developed techniques for proving the correctness of a program. These techniques rely on flow of control in a program being sequential, with each loop having only one entrance and exit.

Although it is possible to construct a program using only if-then-else and while statements, to do so may necessitate an increase in the length of the program. Hence developments of these statements such

as repeat-until and case statements may be introduced. These facilitate the programming of certain control paths without destroying the essential "top-to-bottom" program execution flow.

4.2 Influences on the Design of SUILVEN

In this section, the design criteria on which SUILVEN is based are described, along with some general comments on design decisions which were made at an early stage of the project.

The design criteria which we established for SUILVEN were:-

(i) The language should be a high-level language which should not contain features dependent on the architecture of a particular computer.

(ii) The language should be such that an accurate and readable machine description may be specified.

(iii) The microcode generated by the SUILVEN compiler should execute reasonably efficiently.

(iv) The compiler for SUILVEN should be fairly simple to construct.

Inevitably, with such criteria, conflicts arose and compromises had to be made between conflicting demands. In particular, the need for microcode efficiency often conflicted with features which would have improved the power of SUILVEN as a machine description language.

When such conflicts arose, we generally chose the most efficient

method of solution. Current microprogramming practice emphasises efficiency and, if a high-level language is to be accepted by present users, the code produced must be comparable with machine language in efficiency.

The most significant influence this had on the design of SUILVEN was the adoption of a static rather than a dynamic storage allocation scheme. We believe that a dynamic allocation scheme is not only more general but is also aesthetically preferable to a static scheme. However, the loss of efficiency with such a scheme and our anticipation that features such as recursion and dynamic arrays would not be necessary in implementing s-machines caused a static allocation scheme to be adopted for SUILVEN.

Other design decisions made at an early stage were:-

(i) Only sequential control constructs such as if-then-else and while-do should be included in the language.

(ii) SUILVEN variables should not be typed.

The reasoning behind the former decision has been discussed previously in this chapter. We wholeheartedly agree with the opinion that uncontrolled branching in a program is undesirable and only a simple set of control constructs have been included in SUILVEN.

The decision not to include typed variables is perhaps more controversial. The inclusion of types permits the detection of a number of programming errors at compile-time. Whilst languages such as ALGOL60 have basic types such as integer and boolean, Wirth's PASCAL(WI) has carried typing further and allows

the user to define his own types.

However, for certain applications, the typing of language variables is a distinct disadvantage. If a number of different operations such as real addition, integer multiplication, and logical oring are to be carried out on a storage area(a typical example is a stack), typing that storage area is inconvenient. Either type transfer functions have to built into the language(as in ALGOLW), or such an area must be represented as a PASCAL-type tagged record.

We anticipated that situations where different kinds of operation are carried out on a data area are common in s-machines, and subsequent experiment has shown this to be true. Hence a decision was made to exclude types from SUILVEN.

Individual features of SUILVEN are described in some detail below in an informal manner. The reader is referred to Appendix 1 for a more formal description of the language. As well as describing language features, we also discuss, where appropriate, the reasoning behind the inclusion of certain features in the language.

4.3 The Structure of a SUILVEN Program

A SUILVEN program consists of a sequence of declarations, followed by a sequence of SUILVEN statements. A program is terminated by the reserved word ENDPROGRAM.

SUILVEN declarations include data declarations, structure declarations and procedure declarations. They allow the user to specify and ascribe structure to machine data areas and to give names to machine operations.

SUILVEN statements include assignments, control statements, and procedure calls. They specify operations to be executed using the s-machine data areas.

4.4 SUILVEN Declarations

The declaration part of a SUILVEN program has a threefold purpose:-

(i) To allow the user to define names referring to data areas and sections of SUILVEN code.

(ii) To provide information about these data areas and code sections which the compiler may use in generating microcode.

(iii) To provide a description of the size and structure of the data areas in the s-machine described by the SUILVEN program.

In order to fulfil these purposes, there are five kinds of declaration allowed in a SUILVEN program. The possible declarations are:-

(i) Macro declarations

(ii) Data area declarations

(iii) Structure declarations

(iv) Flag declarations

(v) Procedure and function declarations

These different declarations are covered in turn below.

4.4.1 Macro Declarations

SUILVEN macro declarations are designed to offer facilities for a limited degree of textual replacement within a program. If carefully used, this can improve the readability and ease the maintenance of a program.

A macro is declared:-

```
MACRO  '<string>' = '<string>'
```

and specifies that each occurrence of the string to the left of the equals sign is to be replaced by the string to the right of the equals sign. Notice that only simple substitution is allowed and that it is not possible to pass parameters to a macro.

Macros were included in SUILVEN to allow the naming of constants and the naming of frequently used code segments. For the former purpose the present construct is quite adequate.

However, after using SUILVEN, we now believe that the naming of constants and the naming of code segments should be separate. For the latter purpose, it is desirable that parameter passing should be permitted and any future development of SUILVEN will include this extension to the language.

4.4.2 Data Area Declarations

SUILVEN data area declarations permit the programmer to name and to specify the size of data areas in the abstract machine which he is constructing. The width of each data area is specified exactly

as a number of bits rather than as multiples of some arbitrary cell size, such as a byte.

The declaration setting up these data areas may take the forms:-

(i) BITS(<width>) <identifier list>

(ii) BITS(<width>) ARRAY(<size>) <identifier list>

Declaration (i) above, sets up scalar data elements of the specified width, whereas declaration (ii) sets up vectors of data elements. Examples of each type of declaration are:-

(i) BITS(24) AREG,BREG

This defines two data areas called AREG and BREG each 24 bits wide.

(ii) BITS(32) ARRAY(512) STACK

This defines a data area called STACK, which is a vector of 512 elements each 32 bits wide.

The flexibility allowed in defining the sizes of variables means that the data architecture of an abstract machine may be exactly specified. Space is conserved by eliminating the need to accomodate registers of varying widths in arbitrarily sized storage cells. Similiarly, machine store and storage areas such as stacks may be defined to accomodate data to any precision.

This flexible means of defining data areas is a significant contributor to the descriptive power of SUILVEN and to the efficient implementation of abstract machines.

4.4.3 Structure Declarations

Declarations are provided in SUILVEN which allow the programmer to define the form of a data structure and to declare that a machine data area should be associated with that structure.

This is accomplished by means of a TEMPLATE declaration which defines a template describing the data structure. This template may be assigned to a data area via a DEFINE declaration.

A TEMPLATE declaration has the following syntax:-

```
TEMPLATE <identifier> = <identifier>(<number>)
                        [,<identifier>(<number>)]*
```

The starred square brackets $[]^*$ indicate that the elements enclosed in these brackets may be repeated zero or more times.

A TEMPLATE declaration defines the name of a template consisting of one or more fields of a specified size. For example:-

```
TEMPLATE  LIST_ELEMENT = GC(1),HEAD(24),TAIL(12)
```

This specifies that the template names LIST_ELEMENT may be regarded as having the following fields:-

(i) A 1-bit field named GC

(ii) A 24-bit field named HEAD

(iii) A 12 bit field named TAIL

Templates do not, themselves, define any data areas but merely define a pattern which may be subsequently assigned to a data area. This is accomplished using a DEFINE declaration. This has the form:-

```
DEFINE <template name> : <identifier list>
```

This specifies that the variables named in the identifier list take the structure defined by the named template.

Not only may scalar variables be structured in this way, but also a template may be assigned to a vector. In this case, each element of the array is considered to be of the structure defined by the template.

The utility of these declarations is illustrated by the example declaration sequence shown below:-

```
BITS(48) AREG,BREG
BITS(48) ARRAY(512) STACK
TEMPLATE B5700_WORD = FLAG(1),SIGNM(1),SIGNC(1),MANTISSA(6),
                      CHAR(39)
DEFINE B5700_WORD : AREG, BREG, STACK
```

The above declarations state that the variables AREG, BREG, and STACK are to be considered as structured data areas, having the structure defined by the TEMPLATE declaration. Notice that the declared width of the data areas and the total number of bits declared as components of template fields should be the same.

This type of data structuring was included in SUILVEN as it is often necessary to consider a machine register as being composed of several fields. The example above defines the structure of the stack and the stack registers in a Burrough's B5700 computer. Similiarly, the template for an IBM S/360 program status word would be:-

```
TEMPLATE PSW = SYSMASK(8),PROT_KEY(4),AMWP(4),INTERRUPT(16),
               ILC(2),CC(2),PROGMASK(4),ADDRESS(24)
```

Allowing the user to name the fields within a data area and to access these fields via the names rather than via shift and mask operations reduces the chances of errors occurring when using these fields. In addition, this extra level of abstraction makes for clearer,more readable abstract machine descriptions.

4.4.4 Flag Declarations

The SUILVEN flag declaration facility permits the user to define names which are single-bit logical elements. The need for this type of declaration in a high-level microprogramming language arises for two reasons:-

(i) Within the underlying 'hard' machine on which SUILVEN is implemented there are a number of single-bit flip-flops which detect various interrupt conditions. As it is necessary that the soft machine tests these conditions, a means of so doing must be introduced into the abstract machine programming language. In SUILVEN, these interrupt flip-flops may be accessed via predeclared flags.

(ii) As well as 'hard' interrupts, the soft machine programmer must be able to define flip-flops which can be used to mark soft machine interrupts such as a stack overflow. BITS variables are somewhat clumsy for this function, hence, flag type variables were introduced into SUILVEN.

A FLAG declaration has the syntax:-

FLAG <identifier list>

This merely declares the names of flags which may be accessed using special flag operations which allow flags to be tested and set. These operations are discussed in section 4.5.2, which deals with SUILVEN logical expressions and in section 4.6.1 which covers assignments.

4.4.5 Procedure Declarations

The advantages of including a facility in a programming language which allows code sections to be named and subsequently activated are well known. Procedures allow code to be shared and programs to be structured as a set of logically distinct modules.

Procedures in SUILVEN may be of two kinds:-

(i) Proper procedures, identified by the reserved word PROCEDURE. These procedures are simply named sections of code which may or may not have parameters.

(ii) Function procedures, identified by the reserved word FUNCTION. A function always returns a value and is called as a primary element of an arithmetic expression.

Parameters, in the form of BITS variables, may be passed to a SUILVEN procedure and must be declared as formal parameters in the procedure heading. The syntax of procedure declarations is rather lengthy and, as the concept is familiar, procedure headings are illustrated by example below:-

(i) FUNCTION POPSTACK()

(ii) PROCEDURE PUSH(BITS(4) TAG; BITS(24) VAL)

Declaration (i) above declares a function procedure called POPSTACK. POPSTACK has no formal parameters. Declaration (ii), on the other hand, defines a proper procedure. This is named PUSH and has two formal parameters - a 4-bit variable called TAG and a 24-bit variable called VAL.

The body of a SUILVEN procedure consists of local variable declarations, if required, and statements comprising the code of the procedure. A procedure declaration is terminated by the reserved word END.

The value returned from a function procedure is indicated by following end with an arithmetic expression, where appropriate. For example:-

END = <expression>

The value of that arithmetic expression constitutes the value returned by the function.

4.4.6 Local Declarations

Within a SUILVEN procedure, it is possible to declare names which are local to that procedure. The declarations which may be used to establish local names are:-

(i) Macro declarations

(ii) Data area declarations

(iii) Structure declarations

(iv) Flag declarations

It is not possible to declare procedures which are local to other procedures. This is in keeping with our design decision to use a static rather than a dynamic storage allocation scheme in SUILVEN.

The form of the local declarations for macros, bits variables, and flags is identical to that of the global variable declarations described above. However, local structure declarations have an additional facility(the REDIMENSION declaration) which is only meaningful within procedures. The structure of data areas may be reallocated within a procedure using locally or globally defined templates. This structure modification is covered in more detail below.

4.4.7 Redefining the Structure of Data Areas

It is often desirable to modify the structure of a data area depending on the operation acting on that area. For example, the top element of a stack may be of integer type for an ADD operation yet may take the form of a string descriptor for a CONCATENATE operation.

Within a procedure, structured BITS variables may have their structure redefined using a template which is declared either locally or globally. Naturally, locally declared variables may also be structured using a local or global template.

The example below illustrates the operation and utility of this feature. Consider a global variable defined as follows:-

```
BITS(32) REGISTER
```

For integer arithmetic operations, it may be convenient to regard this register as being made up of a sign bit plus an integer. Hence within the procedure which performs integer arithmetic there might be the following structure declarations:-

```
TEMPLATE INTEGER = SIGN_BIT(1),NUMB(31)
DEFINE INTEGER : REGISTER
```

For other operations such as as conversion from character to integer it may be convenient to regard this register as being composed of four 8-bit characters. Therefore, within a procedure which executes character operations the declarations below might be used:-

```
TEMPLATE CHARS = C1(8),C2(8),C3(8),C4(8)
DEFINE CHARS : REGISTER
```

This ability to locally redefine the structure of variables contributes significantly to the clarity of machine descriptions. In addition, the possibility of error when operating on data areas is reduced, as these areas may be structured according to the function operating on them.

4.4.8 The REDIMENSION Declaration

A fairly common feature shared by a number of machines is to have a range of widths for machine instructions. For example, the IBM S/360 series has certain instructions 2 bytes wide, others 4 bytes wide and a third category 6 bytes wide. Where Huffman encoding techniques, as described in chapter 2, are used it is necessary to utilise instructions of differing widths. Hence, an s-machine implementation language requires some facility to handle differing width instructions.

The SUILVEN REDIMENSION declaration is a declaration which enables the user to redefine his basic cell size in any array. Naturally, by so doing the number of elements in the array is also changed. The syntax of the REDIMENSION declaration is:-

REDIMENSION <array name>(<cell size>,<number of cells>)
[,<array name>(<cell size>,<number of cells>)]*

The declaration specifies that the named arrays should be redimensioned according to the bracketed specifications.

The REDIMENSION declaration enables the soft machine store to be modified in such a way that, irrespective of the width of the instruction, the instruction may be fetched from store in a single operation rather than in a loop. This is perhaps best illustrated using examples.

Example 1

The IBM S/360 series of computers has a byte oriented store. There are four basic types of machine instruction and three different instruction widths - 2 bytes, 4 bytes, and 6 bytes. The REDIMENSION declaration can be illustrated using two of these instruction types:-

(i) The RR type instruction, whose op code occupies one byte and whose parameters occupy one byte.

(ii) The RX type instruction, whose op code occupies one byte and whose parameters occupy three bytes.

The 360 machine store may be declared as a BITS array:-

```
BITS(8) ARRAY(storesize) STORE
```

The procedure for handling RR type instructions would not need to redimension STORE as the instruction parameters are contained in a single byte. For example:-

```
BITS(8) PARAMETERS
.
.
.
PARAMETERS := STORE(.POINTER.)
```

However, the procedure processing RX type instructions must fetch three bytes from STORE to obtain instruction parameters and would therefore redimension STORE to the appropriate format.

```
BITS(24) PARAMETERS
REDIMENSION  STORE(24,size2)
.
.
.
PARAMETERS := STORE(.POINTER.)
.
.
.
```

As the store has been redimensioned, the assignment statement which fetches one cell of the array takes 24 bits from STORE rather than 8 bits.

Example 2

To illustrate how the REDIMENSION statement can be used to handle the situation where Huffman encoding techniques are used to optimise the use of store, we present the example below. Machine codes are encoded in a number of bits inversely proportional to the frequency of instruction occurrence.

Consider the situation where op codes may be encoded in either 3, 7, or 10 bits. Instruction parameters may also be encoded in a varying number of bits. The machine store may be defined as an array of single bits:-

```
BITS(1) ARRAY(storesize)  STORE
```

Defining the machine store in this way allows maximum flexibility when the store is to be regarded as cells of varying size.

Within the procedure which fetches instructions having 3-bit op codes, STORE might be redimensioned:-

```
REDIMENSION  STORE(3,size2)
```

Three instruction bits are always fetched by the instruction decoder and an escape op code signifies whether further bits are to be fetched to make up a 7-bit or a 10-bit code.

The procedure which builds 7-bit opcodes requires 4 additional bits. Store might therefore be redimensioned:-

```
REDIMENSION  STORE(4,size3)
```

Similiarly, the procedure which builds 10 bit codes requires 7 additional bits, and it might redimension store:-

```
REDIMENSION  STORE(7,size4)
```

The REDIMENSION declaration, therefore, provides SUILVEN with a feature allowing the flexible structuring of abstract machine data areas. By using this statement to alter the dimensions of an array, the efficiency of the generated s-machine is increased. This is due to a reduction in the number of store fetch operations required when transferring instructions from store into a register.

Obviously, the REDIMENSION declaration is logically unnecessary but this is a construct which was introduced into SUILVEN in order to increase the efficiency of the generated microcode. As instruction fetching is probably the most frequent s-machine operation we consider the inclusion of the REDIMENSION declaration in SUILVEN to be justified by this potential increase in efficiency

4.5 Expressions in SUILVEN

Before going on to describe the range of SUILVEN statements, it is apposite to describe the form of arithmetic and logical expressions in the language.

As in most high-level languages the expression is the basic element in many language statements. Procedure parameters, the right hand side of assignments, and so on are all represented by expressions. There are two types of expressions in SUILVEN:-

(i) Bits expressions, which are evaluated to produce a bitstring representing some s-machine data type e.g. an integer.

(ii) Logical expressions, which are evaluated to produce a truth value, true or false.

4.5.1 Bits Expressions

SUILVEN bits expressions consist of a sequence of one or more primary quantities possibly separated by operators. The syntax for expressions is fairly simple:-

```
<bits expression> ::= <primary>[<operator><primary>]*

<primary>         ::= <simple variable>|<function designator>|
                      <array element>|<field of structured variable>|
                      <constant>

<operator>        ::= + | - | * | / | REM | SHL | SHR | & | XOR
                    | <or>

<or>              ::= |   where | is not a metasymbol
```

Bits expressions are evaluated on a strict left-to-right basis. Bracketing is not allowed and no one operator takes precedence over another.

This very simple method of expression evaluation, rather than the more usual precedence based system, was again a feature of SUILVEN which was promoted for efficiency reasons. The B1700 arithmetic unit is a simple module with no provision for the storage of intermediate results. Therefore, in order to avoid introducing a clumsy scheme for storing intermediate results, it was decided to use a system of expression evaluation which did not produce intermediate results.

In practice, there seems to be little or no need for the evaluation of complicated arithmetic expressions within an s-machine interpreter program. Hence, a simple left-to-right expression evaluation scheme as in SUILVEN causes few practical problems for the s-machine implementor.

The operands in an expression need not all be the same width. Before evaluation all operands in an expression are automatically right justified and left padded with zeros to the size of the widest operand.

Notice that a result of SUILVEN's lack of typing is that arithmetic, shift and logical operations may be intermixed within an expression.

4.5.2 Logical Expressions

SUILVEN logical expressions are used in control statements and are evaluated to yield a value which is either true or false. The form of logical expression syntax is:-

<logical expression> ::=<logical primary>[<connective> <logical primary>]*

<logical primary> ::= <condition>|<flag test>

<condition> ::= <bits expression><relation><bits expression>

<flag test> ::= TRUE(flag name) | FALSE(flag name)

<connective> ::= AND | OR

<relation> ::= = | ≠ | > | < | >= | <=

Logical expressions allow the comparison of bits expressions using the relational operators which have their usual meanings. The connectives AND and OR allow a number of different tests to be carried out in the same expression. For example:-

A = B AND C = D OR P = Q

Flag tests, which may also be used as logical primaries, allow the SUILVEN programmer to test the value of a flag variable. The tests have their obvious meaning.

4.6 SUILVEN Statements

This section of this chapter on SUILVEN covers the possible statements available to the SUILVEN programmer. These statements fall into three distinct classes:-

(i) Assignment statements

(ii) Procedure calls

(iii) Control statements

Statements may be combined into compound statements by enclosing them in tagged brackets £(and £). The sections below deal with each type of SUILVEN statement.

4.6.1 The Assignment Statement

The SUILVEN assignment statement is similiar in syntax and semantics to the assignment statement in most other high-level ALGOL-like languages. Its syntax is:-

<LHS> := <bits expression>

where the left hand side may be a simple variable, a field of a structured variable or an indexed variable. Examples of each of these are:-

(i) A := B

A simple variable assignment

(ii) STORE(.POINTER.) := AREGISTER

An assignment to an indexed variable

(iii) LISTR.HEAD := A + B

An assignment to the field(HEAD) of a structured variable

Flag assignment is carried out via a different mechanism. Two standard procedures SET(<flag>) and UNSET(<flag>) are used to assign values of true and false to flag variables.

4.6.2 Procedure Calls

Like the assignment statement, the syntax and semantics of SUILVEN procedure calls is similiar to that of other high-level languages. A procedure is called by writing the name of the procedure followed by a bracketed list of actual parameters(if any). The syntax of procedure calls is defined:-

<procedure call> ::= <procedure name>(<actual parameter list>)

The actual parameters are evaluated and their values are passed to the procedure. Parameter passing by value is the only method possible in SUILVEN. The decision to allow only the values of parameters to be passed to procedures was taken for a number of reasons:-

(i) Safety - there is no way in which the values of global machine data areas may be accidently modified within a procedure.

(ii) Clarity - to modify a global variable it must be referred to by name.

(iii) Efficiency - after parameter values have been copied no further overheads are incurred.

Some examples of procedure calls are:-

(i) GET_NEW_VALUES()

This is a call of the procedure GET_NEW_VALUES which does not take any parameters.

(ii) PUSHSTACK(TREG + SREG)

The procedure PUSHSTACK which has one parameter is called. The expression TREG + SREG is evaluated and its value passed to PUSHSTACK.

4.6.3 SUILVEN Control Statements

As discussed earlier in this chapter(section 4.1.2), recent computer science developments have suggested that the use of certain control constructs such as the if-then-else statement and the while statement produces more understandable and more reliable programs than the use of conditional and unconditional go to statements. This has influenced the design of the control constructs included in SUILVEN with the result that only structured control statements have been included in the language.

Necessary control statements in a programming language are the if-then-else and while-do statements. However, for reasons of efficiency and convenience other control statements have also been included in SUILVEN. These not only include extensions of basic control constructs but also statements which allow the user to directly exit from a procedure and to abort the program.

The set of control statements available in SUILVEN is composed of:-

(i) An if-then-else statement

(ii) A while-do statement

(iii) A case statement

(iv) A repeat-until-do statement

(v) An exit statement

(vi) A stop statement

4.6.4 The If and While Statements

These statements are the well known control statements available in many current high-level languages such as PASCAL and ALGOLW. Their use is illustrated by example:-

Examples of If Statements

(i)
```
IF  A = B  AND  C > D THEN
    £(
        X := Y;  P := Q;
    £)
```

(ii)
```
IF X = 0 THEN
    £(
        SET(ANY_INTERRUPT);  SET(DIVZERO);
    £)
ELSE
    PUSHSTACK(Y/X);
```

Examples of While Statements

```
(i)     WHILE  LIST(.POINT.).TAIL ≠ NIL  DO
           POINT := LIST(.POINT.).TAIL;

(ii)    WHILE  A <= STACKSIZE  DO
           £(
              STACK(.A.) := 0;  A := A + 1;
           £)
```

4.6.5 The Case Statement

The SUILVEN case statement is derived from the case statement of ALGOLW and selects a statement for execution depending on the value of some arithmetic expression. The form of the statement is:-

```
CASE <bits expression> OF
   <statement list>
ENDCASE
```

The arithmetic expression is evaluated and the statement whose position in the list corresponds to that value is selected for execution.

In a language for implementing s-machines, this type of statement is especially useful for selecting an execution sequence on the basis of an instruction op code.

For example:-

```
CASE  OP_CODE  OF
    PUSH(A + B);              "ADD"
    PUSH(A - B);              "SUB"
    PUSH(A * B);              "MUL"
    .
    .
    .
ENDCASE;
```

With this type of case statement, the statement selected for execution has an implicit dependance on the value of the expression used in the case statement. This may be compared with the type of case statement where the relation between the case expression and the statement executed is explicitely stated as in PASCAL. Another alternative form of case statement is the guarded command set recently proposed by Dijkstra(D4).

These types of case statement are safer as they are not susceptible to errors made by the programmer in the ordering of the statement list. This we were aware of when designing the SUILVEN case statement but efficiency reasons again governed our choice of statement. The simpler type of case statement used in SUILVEN can be compiled to more efficient microcode, both in terms of space and execution speed.

In practice, the main function of the case statement in s-machine implementations is to select a statement for execution on the basis of an instruction op code. As these op codes are generally

sequential, without gaps in the op code sequence, the SUILVEN case statement is not seen to be significantly disadvantageous.

4.6.6 The Repeat-Until-Do Statement

The repeat-until-do statement, a modified form of the familiar repeat-until statement, permits the test for loop termination to be placed anywhere within a loop.

The syntax of this statement is:-

<repeat statement> ::= REPEAT <statement> UNTIL <condition>
DO <statement>

The first statement is executed and the condition tested. If this condition is false, the second statement is executed. This process is repeated until the condition is true,when the loop terminates. As either statement may be a null statement this arrangement allows the loop termination test to be positioned at the start, in the middle, or at the end of the loop.

This type of construct obviates the necessity of using boolean variables or go to statements within a loop in order that the test for exit may be made after some other statement in the loop.

A common situation where this occurs is when data is being fetched and processed,with the loop terminating when all data has been consumed. The examples below compare programs where this is handled using boolean variables and where the repeat-until-do statement is used.

Using Boolean Variables

```
SET(NOTFINISHED)
WHILE TRUE(NOTFINISHED) DO
   £(
        GET_DATA();
        IF TRUE(END_DATA) THEN
            UNSET(NOTFINISHED)
        ELSE
            PROCESS_DATA();
   £)
```

Using the Repeat Statement

```
REPEAT
   GET_DATA()
UNTIL
   TRUE(END_DATA)
DO
   PROCESS_DATA();
```

Clearly the repeat statement is more natural and concise. It also has the advantage that more efficient code may be generated as only one rather than two tests need be made on each loop execution.

4.6.7 The Exit and Stop Statements

The exit and stop statements respectively allow the programmer to leave a procedure or to terminate the program. They are represented by the reserved words EXIT and STOP.

The inclusion of an exit statement allows the return from a procedure if some exceptional condition is encountered. Naturally, this may be achieved using nested if-then-else statements but an explicit exit is often clearer and more efficient.

A stop statement was introduced into the language as it is often necessary to terminate the program on the detection of an unrecoverable error. Again, a possibly complex sequence of if-then-else statements can be avoided without loss of clarity.

4.7 Using SUILVEN to Describe an IBM S/360 Computer

Perhaps the best way to illustrate a programming language is to present an example of a familiar problem in that language. The example below shows part of the SUILVEN description of a machine in the IBM S/360 range, whose architecture is well known. Naturally, a complete description would be very lengthy and, as a result, only the data area description and descriptions of representative instructions are shown below.

An IBM S/360 computer has a 'general purpose' architecture. Its store is made up of 8-bit bytes grouped into 4-byte words, it has 16 general purpose registers with program control information stored in a program status word. It has a comprehensive instruction

set which includes register-register, register-store, and store-store instructions. A full description of the machine may be found in the appropriate IBM reference manual(I4).

The example below describes the machine data areas, the Add Register instruction(an RR instruction), the Compare instruction(an RX instruction), and the Move Character instruction(an SS instruction). All 360 op codes occupy one byte and the instruction descriptions below assume this has been identified. For clarity, comments in the machine description are included in italic type, although this is not a feature of SUILVEN.

```
" This is an example program in SUILVEN describing part of
  an IBM S/360 computer. Data areas visible and invisible to
  the machine language programmer are described along with
  the operation of instructions representative of each
  instruction type.
  Firstly, describe the machine data areas accessible to the
  programmer                                                  "

BITS(32) ARRAY(16)  REGISTERS;
BITS(8) ARRAY(STORESIZE)  STORE;
BITS(64) PROGRAM_STATUS_WORD;

"  Define the structure of the PSW and assign it    "

TEMPLATE  PSW = SYSMASK(8), PROTKEY(4), AMWPC(4), INTCODE(16),
                ILC(2), CC(2), PROGMASK(4), ADDRESS(24);
DEFINE  PSW : PROGRAM_STATUS_WORD;
```

```
"   Now define some internal registers used by the machine
    in instruction execution    "

BITS(4)  SR, DR, IR;
BITS(12) DISP, DISP2;
BITS(8)  OPCODE, SSLEN;

"   Define some utility procedures.  These are procedures used
    by a number of instructions to perform tasks which are common
    to each    "

PROCEDURE  GET_RR_PARAMETERS;

    "   Fetches the parameters for an RR type instruction.  The
        source register is left in SR and the destination
        register in DR    "

    BITS(8) P;
    TEMPLATE RR = DR(4), SR(4);
    DEFINE  RR : P;

    "   Fetch instruction parameter into P.  Assume that the
        ADDRESS field of the PSW points to this    "

    P := STORE(.PROGRAM_STATUS_WORD.ADDRESS.);
    PROGRAM_STATUS_WORD.ADDRESS := PROGRAM_STATUS_WORD.ADDRESS + 1;
    SR := P.SR;          DR := P.DR;
END;
```

```
PROCEDURE GET_RX_PARAMETERS;

    " Fetches the parameters for an RX type instruction. The
      destination register is left in DR, the index register in
      IR, the base register in SR and the displacement in DISP.
      As the parameters for an RX instruction occupy 24 bits
      STORE is redimensioned so that all 24 bits may be fetched
      in a single instruction "

    REDIMENSION STORE(24,STORESIZE2);
    TEMPLATE RX = DR(4),IR(4),BR(4),DISP(12);
    BITS(24) P, SP;
    DEFINE RX : P;

    " SP is a pointer into the redimensioned store area. Compute
      its value from the PSW ADDRESS field "

    SP := PROGRAM_STATUS_WORD.ADDRESS / 3;
    PROGRAM_STATUS_WORD.ADDRESS := PROGRAM_STATUS_WORD.ADDRESS + 3;
    P := STORE(.SP.);
    DR := P.DR;    IR := P.IR;    SR := P.BR;    DISP := P.DISP;
END;

PROCEDURE GET_SS_PARAMETERS;

" As for above procedures but for SS type instructions. The
  number of bytes involved in the instruction is in SSLEN,
  the base registers are DR and SR, and the displacements are
  in DISP and DISP2. Store is again redimensioned, this time
  to a cell size of 40 "
```

```
    BITS(40)  P;     BITS(24)  SP;
    TEMPLATE SS = LEN(8), BR1(4),DISP1(12), BR2(4), DISP2(12);
    DEFINE  SS : P;
    REDIMENSION  STORE(40,STORESIZE3);

    "  Compute value of store pointer(SP) from PSW ADDRESS field  "

    SP := PROGRAM_STATUS_WORD.ADDRESS / 5;
    PROGRAM_STATUS_WORD.ADDRESS := PROGRAM_STATUS_WORD.ADDRESS + 5;
    P := STORE(.SP.);
    SSLEN := P.LEN;     SR := P.BR1;     DISP := P.DISP1;
    DR := P.BR2;     DISP2 := P.DISP2;
END;

PROCEDURE ADD_REGISTER:

    "  This is an RR instruction which adds the values in the
       specified registers.  The condition code field of the
       PSW is set by this instruction     "

    GET_RR_PARAMETERS;
    REGISTERS(.DR.) := REGISTERS(.DR.) + REGISTERS(.SR.);

    "  Assume the existence of a flag variable OFLOW which is set
       if integer overflow occurs.  OFLOW would be an internal
       machine flip-flop associated with the adder     "

    IF  TRUE(OFLOW) THEN
        PROGRAM_STATUS_WORD.CC := 3
    ELSE
```

```
        IF REGISTERS(.DR.) > 0 THEN
            PROGRAM_STATUS_WORD.CC := 2
        ELSE
            IF REGISTERS(.DR.) = 0 THEN
                PROGRAM_STATUS_WORD.CC := 0
            ELSE
                PROGRAM_STATUS_WORD.CC := 1;
END;

PROCEDURE COMPARE;

    " Compare is an RX type instruction which compares the
      value in a register with a value held in store. The
      condition code field of the PSW is set on the basis of
      this comparison.
      A temporary register TEMP is used in this instruction
      to hold the value of the operand in store. This avoids
      unnecessary store fetches "

    BITS(32)  TEMP;
    GET_RX_PARAMETERS;
    IF  IR = 0 THEN
        TEMP := STORE(.REGISTERS(.SR.) + DISP.)
    ELSE
        TEMP := STORE(.REGISTERS(.SR.) + REGISTERS(.IR.) + DISP.);
    IF  REGISTERS(.DR.)   TEMP  THEN
        PROGRAM_STATUS_WORD.CC := 2
    ELSE
```

```
        IF  REGISTERS(.DR.) = TEMP  THEN
            PROGRAM_STATUS_WORD.CC := 0
        ELSE
            PROGRAM_STATUS_WORD := 1;
END;

PROCEDURE MOVE_CHARACTERS;
```

" *This is an example of an SS type instruction. The function of the instruction is to move a specified number of characters(bytes) from one location in store to another The instruction uses some temporary registers to hold intermediate addresses and lengths.* "

```
    BITS(24)  ADDR, ADDR2;          BITS(8)  COUNTER;
    GET_SS_PARAMETERS;
    COUNTER := 0;
    ADDR := REGISTERS(.SR.) + DISP;
    ADDR2 := REGISTERS(.DR.) + DISP2;
    REPEAT
        STORE(.ADDR1.) := STORE(.ADDR2.)
    UNTIL
        COUNTER = SSLEN - 1
    DO
        £(  ADDR := ADDR + 1;
            ADDR2 := ADDR2 + 1;
            COUNTER := COUNTER + 1;      £)
END;
```

The above description gives some idea how SUILVEN may be used to describe a computer such as the IBM S/360. However, it must be borne in mind that this is a description of a 'hard' machine. SUILVEN is designed primarily for the description of language-oriented soft machines and is therefore not ideal for describing machines like the S/360.

4.8 Summary and Conclusions

This chapter has provided a description of the design of a programming language called SUILVEN. This language is intended for the description and implementation of language-oriented soft machines.

SUILVEN contains powerful data description features enabling the user to express the full range of data organisations which might occur in abstract machines. These data description facilities include declarations for defining the width of data areas and declarations for assigning structure to these areas.

The statements of SUILVEN are made up of assignments, procedure calls and control statements. Only a limited but adequate set of control constructs have been provided and the go to statement has been completely excluded from SUILVEN. To facilitate loop exit a repeat-until-do statement has been included in the language. This statement permits the test for loop exit to be placed at the beginning, the middle, or the end of the loop.

The reader will have noticed that no input/output statements have been described in this chapter. No such statements are defined in SUILVEN. This was a deliberate decision made for the following reasons:-

(i) We were unsure of the I/O requirements of s-machines. The s-machines which were studied all carried out I/O via calls to the underlying operating system.

(ii) SUILVEN was originally intended to produce microprograms which would run on our B1700 simulator. As this had minimal I/O facilities it was decided to postpone the decision on which I/O facilities were to be included in SUILVEN.

Simple input/output has now been implemented as standard procedures in the language. These are described in the following chapter which covers the implementation of SUILVEN.

SUILVEN has now been used to implement a number of different abstract machines. Using these implementations, the language design has been evaluated as a description language and as an implementation language. A full description of this evaluation is given in chapter 7 and it is summarised below.

In general terms, the design criteria for SUILVEN which were established in section 4.2 have been fulfilled. The language is a high-level machine description language without machine dependent features. It is, however, oriented towards the B1700 computer series. The SUILVEN compiler was reasonably easy to implement.

It is in criterion (iii), i.e. that the microcode generated from a SUILVEN program be efficient, that SUILVEN is not as satisfactory as may be desired. Compromises have been made to make the language more efficient, but there are still serious inefficiencies in the microcode generated for certain constructs.

We now believe that the requirements of a machine description language and a high-level microprogramming language are different, and that the inefficiencies of SUILVEN are a consequence of attempting to combine these functions. This will be discussed in more detail in chapter 7.

CHAPTER 5

THE IMPLEMENTATION OF SUILVEN

This chapter describes the implementation of a compiler for SUILVEN. The SUILVEN compiler is programmed in SNOBOL4 and uses a top-down recursive descent syntax analysis technique to parse SUILVEN statements. The intermediate form of B1700 microcode, which is described in chapter 3, is generated by the compiler.

Compiler construction has been well documented by authors such as Randell and Russell(R1) and Gries(G2). Therefore, a full description of the compilation of each language feature is not given here. Instead, the design decisions involved in the implementation of the compiler are discussed, along with an overview of the compilation process. The implementation of SUILVEN's data description features is described in some detail as these constructs are not available in other programming languages and hence their compilation is not widely documented.

The chapter concludes with a discussion of the problems involved in optimising SUILVEN code and presents some information on the performance of the SUILVEN compiler.

5.1 Compiler Design

The overall structure of a compiler program is well known. It consists of a number of modules performing different functions associated with the translation process.

These functions can be categorised as follows:-

(i) Lexical analysis

(ii) Syntactic and semantic analysis

(iii) Code generation

In addition to these essential modules, the compiler may include a module designed to optimise the generated code.

Compilers may make one or more passes through the source code being translated. In a single-pass compiler, all the above functions are carried out in a single pass through the code being compiled, whereas in a multi-pass compiler, several passes are made through the code. Each of these passes often corresponds to one of the above functions with an intermediate form of the source code being generated by each pass.

The lexical analysis phase of a compiler is generally straightforward and consists merely of identifying tokens from the input source code. These tokens may be entered in a symbol table, if appropriate, and a numeric code representing the token passed to the syntax analysis phase of the compiler.

The generation of compiled code is still essentially an *ad hoc* process. Although intermediate code forms such as reverse Polish notation or triples may be produced, the generation of code is still largely dependent on the intimate machine knowledge of the compiler writer.

Syntax analysis, however, is much less of an *ad hoc* process and a good deal of research has been carried out in this field, investigating different analysis techniques. Fundamentally, a programming language syntax may be analysed in either a top-down or a bottom-up manner.

Using bottom-up techniques, the source string is reduced until the goal symbol of the language grammar is attained. The reduction of phrases in the language depends on the precedence of language symbols, and syntax analysis methods depending on simple precedence, operator precedence, and mixed-strategy precedence(W6,F2,M1) have been developed.

Top-down syntax analysis methods depend on starting with the goal symbol of the language grammar and deriving the source string from this.

Probably the most straightforward technique of top-down analysis is the method of recursive descent. This method represents each non-terminal symbol in the language grammar by a recursive procedure which checks the syntax of that particular non-terminal. A syntax tree is therefore constructed implicitely by a sequence of procedure calls.

As an example of the simplicity and elegance of the recursive descent analysis technique, the example below presents a procedure called WHILE_STATEMENT. This procedure parses the non-terminal <while statement> in a language grammar.

```
WHILE_STATEMENT
   BEGIN
        CONDITIONAL_EXPRESSION
        NEXT_SYMBOL
        CHECK("DO")
        NEXT_SYMBOL
        STATEMENT
   END
```

When a recursive descent analysis technique is used, only one or two symbols are required from the input stream at any one time. Semantic analysis and code generation are generally intermixed with syntax analysis. Recursive descent compilers, therefore, are basically one-pass compilers, although extra passes may be included to optimise the generated code.

Recursive descent compilers are simple to construct, efficient, and easy to understand. However, the class of languages compilable via this technique is restricted by the following conditions:-

(i) All names used in a program must be declared before they are used in program statements.

(ii) The language grammar must be LL(1). This means that parsing decisions can be made on the basis of the current symbol from the input stream which is available to the compiler.

These requirements are not usually disadvantageous, especially in the situation where the language design and compiler construction proceed in parallel. Simple language modifications are then possible in order that the language be made compilable using a recursive descent technique. Under special circumstances, it may not be possible to modify the language and, in this case, it is convenient to 'cheat' by looking ahead in order to make a parsing decision. Such circumstances are rare and do not impose significant compiler overhead.

5.2 The Compiler Programming Language

The choice of a language in which to program a system is one which has to be made at an early stage of the system design. Features available in a programming language can simplify certain implementation techniques, whereas the lack of particular features can render some techniques completely impractical.

There were a number of programming languages available to us. These fell into three categories:-

(i) Low-level languages(S/360 Assembler, PL360)

(ii) General-purpose high level languages(FORTRAN, ALGOLW, SNOBOL4, BCPL)

(iii) Special-purpose compiler writing languages(XPL)

The decision which language to use for writing the SUILVEN compiler was governed by a number of factors:-

(i) The suitability of the language for compiler writing. This is primarily governed by the data types and structures available in the language.

(ii) The cost of using a particular language processor on our system. Different language processors require differing amounts of system resources such as disk files and machine store. As program turnround time is a function of the system resources utilised it was to our advantage to minimise resource consumption.

Notice that the efficiency of the compiler for SUILVEN is not of great importance as the language is a research tool,with the compiler used by relatively few people.

Although the low-level language and FORTRAN compilers consumed the least amount of system resources they were immediately rejected as compiler writing tools, because of their unsuitability for this application. Our choice of programming language was therefore narrowed to four alternatives:-

(i) XPL

(ii) ALGOLW

(iii) SNOBOL4

(iv) BCPL

XPL(M1), a PL/1 based language, is part of a translator writing system which includes an automatic precedence table generator and a syntax analyser. This syntax analyser uses a bottom-up mixed-strategy precedence analysis technique.

These features aid the production of a compiler and preliminary experiments were carried out with XPL. Unfortunately, these indicated that the system resources required by the language processor(4 disk files, 200K bytes of store) would make the production of a large program impractical. XPL was therefore rejected as a compiler writing tool.

For similiar reasons, BCPL was rejected as the compiler programming language. Not only did the BCPL compiler require a large amount of system resources but the language also suffered from an unreliable implementation.

Neither ALGOLW or SNOBOL4 suffered from this disadvantage. Both these languages had efficient and reliable implementations and neither consumed an excessive amount of system resources. Our choice, therefore, was dependent on the suitability of each language for compiler construction. Finally, we choose SNOBOL4 as the compiler programming language for the following reasons:-

(i) SNOBOL's inbuilt data types and data structuring features are significantly more powerful than those of ALGOLW. SNOBOL has a primitive data type called TABLE, which can be regarded as an associatively addressed array, and a facility for the user to define his own structured data types. The combination of these features makes the construction and use of compiler tables a relatively trivial task.

(ii) SNOBOL's powerful string manipulation and pattern matching features considerably simplify the production of some parts of the compiler, such as the lexical analyser and the code generator.

(iii) Our ALGOLW implementation imposed restrictions on the maximum number of possible record types. We anticipated that this might cause problems when implementing a compiler.

SNOBOL4 undoubtedly suffers in comparison with ALGOLW inasmuch as it lacks structured control constructs and, in that little compile-time checking is carried out by the SNOBOL4 compiler. In spite of this, because of its data structures and superior string manipulation facilities, SNOBOL4 was chosen as the language for the SUILVEN compiler.

5.3 The SUILVEN Compiler

This section of the thesis presents a brief overview of the SUILVEN compiler and displays, in diagrammatic form, the structure of that program.

Having chosen SNOBOL4 as the compiler programming language it was decided to utilise a top-down recursive descent syntax analysis technique to parse SUILVEN statements. This decision was made for the reasons discussed in section 5.1, viz, recursive descent compilers are simple to construct, efficient, and easy to understand.

There are three basic parts to the SUILVEN compiler:-

(i) The scanner or lexical analyser

(ii) The syntactic and semantic analyser

(iii) The code generator

The scanner and the code generator are relatively simple procedures called by the analysis section as required. Their functions are to return the next symbol from the input stream and output microinstructions respectively. Microinstructions are not output in binary form but in the intermediate form, IML, described in chaper 3.

The syntactic and semantic analyser comprises the major part of the compiler. Its functions are to build the compiler tables from information supplied by SUILVEN declarations, to check the syntax and semantics of SUILVEN statements, and to translate these statements into microcode.

This part of the compiler was constructed in a top-down fashion and has a hierarchical structure. This structure is displayed below in a simplified form in figure 5.1. The nodes of the tree illustrating the program are procedure names whose function is, hopefully, obvious.

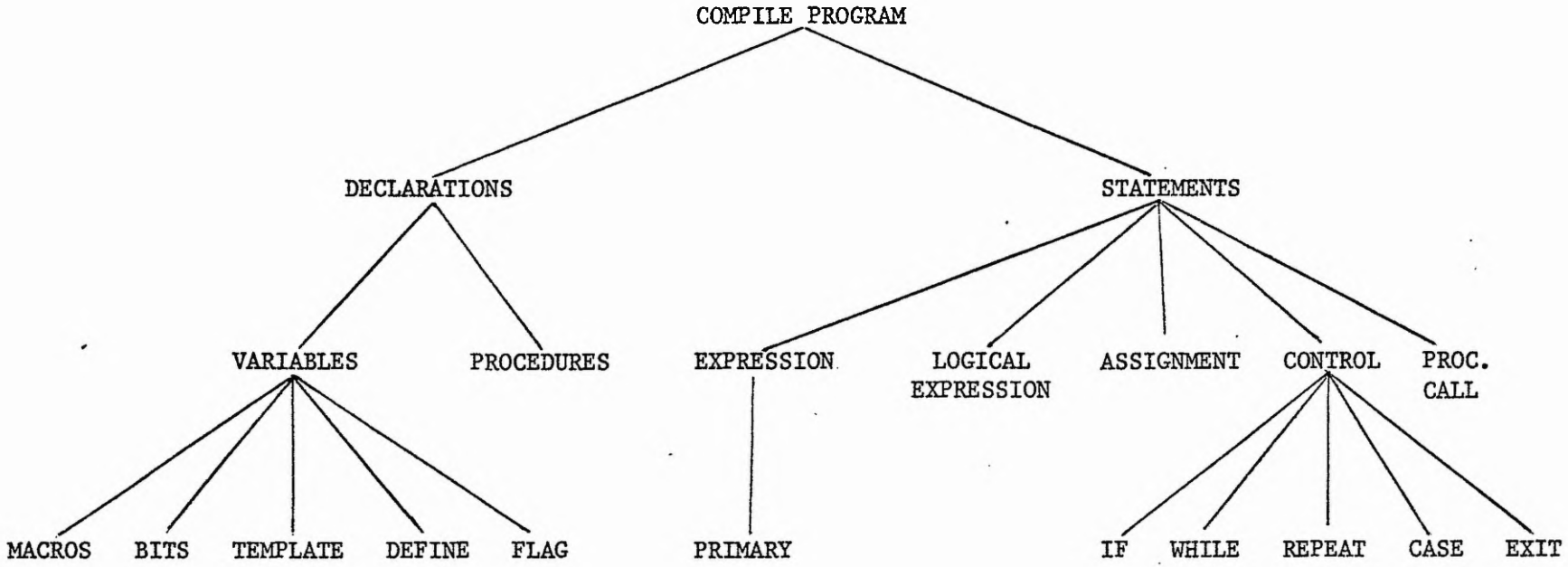


FIGURE 5.1

THE STRUCTURE OF THE SUILVEN COMPILER

The remaining sections of this chapter describe the implementation of SUILVEN's data description features and, very briefly, the translation of SUILVEN statements. The chapter concludes with sections which discuss code optimisation and the performance of the SUILVEN compiler.

5.4 The Implementation of SUILVEN's Data Description Features

In this section, we discuss in some detail the implementation of SUILVEN's data description declarations. These features are covered in some detail as they are the features of SUILVEN which distinguish the language from other high-level languages.

The implementation of SUILVEN declaration requires a number of tables to be built by the compiler to hold information about declared names. The major tables used in the compiler are:-

(i) A symbol table

(ii) A template table

(iii) A procedure table

The structure and functions of each of these tables is described below.

The implementation of all these tables in SNOBOL4 is greatly simplified by SNOBOL's associative addressing feature and data definition facilities. This latter feature allows the structure of a table entry to be defined using a DATA declaration. Any table entry can therefore be made by specifying the table name, the name to be entered, and a list of attributes.

For example:-

```
SYMBOLS<'STACK'> = SYMBOLENTRY(24,ARRAYTYPE,50,2000,0,NIL)
```

This would enter the name STACK into the table SYMBOLS along with the attributes of STACK which are specified in brackets. SYMBOLENTRY is the name given to the structured type representing an entry in the symbol table.

5.4.1 The Symbol Table

The symbol table is the main compiler table. It holds information concerning the data areas declared using the BITS declaration, with each entry structured into six fields as follows:-

(i) LEN

Holds the width, in bits, of a variable if it is a scalar variable or the width of each array element if an array type variable.

(ii) TYPE

Specifies whether the symbol table entry is a scalar, a flag, or an array.

(iii) SIZE

If the entry is an array, specifies the number of elements in the array.

(iv) ADDRESS

This field holds the bit address in machine store of the declared variable if a BITS variable, or, if a FLAG variable, the bit in the L register representing that flag.

(v) SADD

This field specifies whether a variable is in a fast scratchpad register or in store. This is discussed in more detail in section 5.7 which covers microcode optimisation.

(vi) TEMP

If a BITS variable is structured, this field points to the appropriate structure in the template table.

The examples below illustrate typical symbol table entries for a sequence of declarations.

Declarations

```
BITS(24)  AREG,BREG;

BITS(24)  ARRAY (512) STACK;

FLAG   AFULL, BFULL;
```

Symbol Table Entries

Assume that the first bit address allocated is address 2000.

```
AREG    24, S, 0, 2000, 0, NULL

BREG    24, S, 0, 2024, 0, NULL

STACK   24, A, 512, 2048, 0, NULL

AFULL   1, F, 0, 0, 0, NULL

BFULL   1, F, 0, 1, 0, NULL
```

5.4.2 The Template Table

The template table is a table which holds the form of each template declared in the program. For each template, a linked list is constructed with each element in the list holding information about one field of the template. It is necessary to use a list organisation for this table, as a template may have any number of fields.

Each element in the linked list is divided into four fields:-

(i) NAME

The name of the template field.

(ii) LENGTH

The width of the template field in bits.

(iii) OFFSET

The displacement of the field in bits from the beginning of the template structure.

(iv) NEXT

A pointer to the next field of the template.

Figure 5.2, below, illustrates the structure of the template table for typical declarations.

Declarations

```
TEMPLATE  LIST = GC(1),TAG(4),HEAD(20), TAIL(20);

TEMPLATE  STACKSTRUC = TAG(4), DATA(20);
```

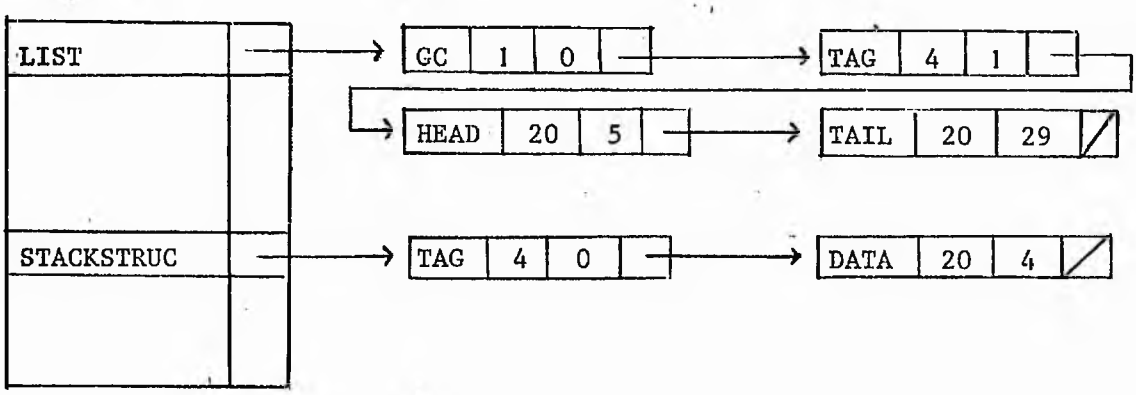


FIGURE 5.2

THE STRUCTURE OF THE TEMPLATE TABLE

A DEFINE declaration causes a template to be associated with a declared program data area. This association is accomplished by linking the appropriate entries in the symbol table and the template table using the TEMP field of the symbol table entry.

Figure 5.3 below illustrates this linking for the following DEFINE declaration:-

```
DEFINE  STACKSTRUC : AREG, BREG, STACK;
```

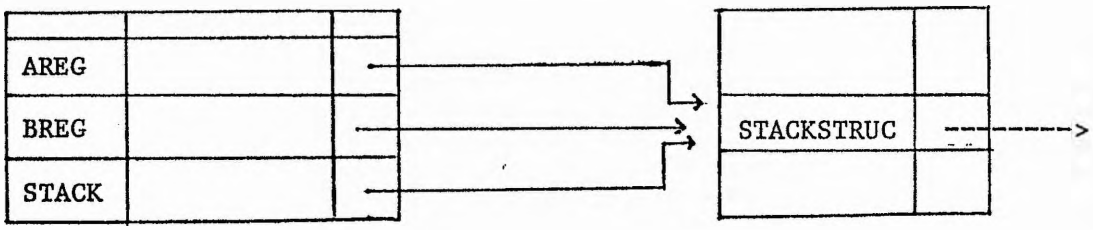


FIGURE 5.3

RELATING A NAME TO A STRUCTURE

5.4.3 Local Declarations

An important contibution to the power of SUILVEN's data description facilities is provided by the ability to define names local to a procedure.

As there are only two levels of scope(local and global) in SUILVEN, local declarations are set up in essentially separate tables. If the local declaration pertains exclusively to locally defined names, the declarations are handled as the global declarations described above.

Recall, however, that it is possible to locally redefine the structure of global data areas either by using a TEMPLATE or by a REDIMENSION statement. This situation is handled by taking a copy of the appropriate global table entry in the local table and associating the locally declared attributes with this copy. The example below illustrates the process.

The global declarations set up 32-bit variables structured as two 16-bit fields:-

```
BITS(32)  ARRAY(250)  STORE;

BITS(32)  REGISTER;

TEMPLATE  HALFWORDS = HW1(16), HW2(16);

DEFINE  HALFWORDS : STORE, REGISTER;
```

Assume that a particular function requires 8-bit variables. The procedure implementing that function could have the declarations:-

```
TEMPLATE  BYTES = B1(8), B2(8), B3(8), B(4);

DEFINE  BYTES : REGISTER;

REDIMENSION  STORE (8,1000);
```

The organisation of the local and global compiler tables after this sequence of declarations has been processed is shown in figure 5.4 below.

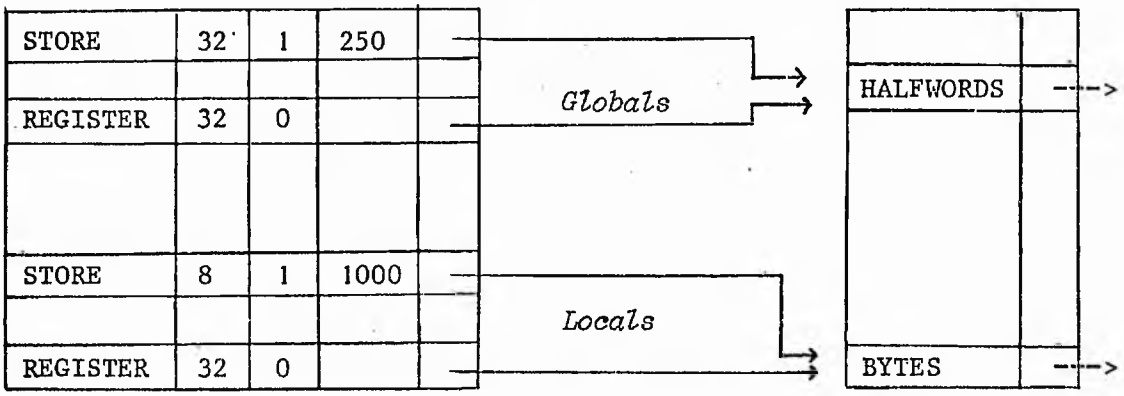


FIGURE 5.4

TABLE ORGANISATION AFTER STRUCTURE REDEFINITION

Naturally, when compiling a SUILVEN procedure, the local table is searched before the global table and this table is re-initialised at the beginning of each procedure header.

5.4.4 The Procedure Table

Information pertaining to SUILVEN procedure declarations is stored in a separate table called PROCEDURE_DICTIONARY. Each entry in this table is structured into five fields as follows:-

(i) NOFORMALS

The number of formal parameters.

(ii) PTYPE

Indicates whether the procedure is a function or a proper procedure.

(iii) PADDR

The control store address of the first microinstruction in the procedure.

(iv) FADDR

The address of the first formal parameter.

(v) FPLEN

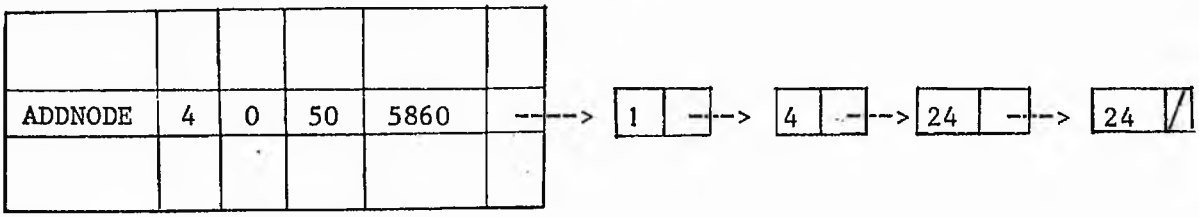
A pointer to a linked list holding the width of each formal parameter.

Unlike fixed word length systems, it is not enough to hold the number of formal parameters and the address of the first parameter. Because of the possibility of formal parameters having different widths, it is necessary to save the width of each. This is accomplished using a linked list.

Again, the structure of an entry in the procedure dictionary is best illustrated by example. Consider the procedure head:-

```
PROCEDURE  ADDNODE(BITS(1) GC; BITS(4) TAG;
                                BITS(24)  HEAD,TAIL );
```

Figure 5.5 below shows the form of the entry in PROCEDURE_DICTIONARY for this declaration.



PROCEDURE_DICTIONARY

FIGURE 5.5

AN ENTRY IN PROCEDURE DICTIONARY

5.5 Compiling SUILVEN Statements

The compilation of language statements such as assignment, procedure calls, and control statements is admirably documented by authors such as Gries(G2), and Randell and Russell(R1). The general techniques described therin have been followed in the implementation of the SUILVEN compiler.

A repetition of the description of how to compile such language statements would be tedious, especially as these techniques are well known. We therefore confine ourselves to a brief description of the compiler procedure STATEMENTS which is the main procedure used in the compilation of SUILVEN statements.

SUILVEN control statements may all be identified without lookahead by the reserved word beginning the statement. Assignment statements and procedure calls are distinguished using a single symbol lokkahead. If the next symbol is a left parenthesis the compiler assumes the statement is a procedure call, otherwise the statement is taken as an assignment statement. This is the only example of compiler lookahead used.

The structure of the procedure STATEMENTS is shown below:-

```
case SYMBOL of

    "IF" : IF_STATEMENT;
    "WHILE" : WHILE_STATEMENT;
              .
              .
              .

endcase
    else
        if LOOKAHEAD = "(" then
            PROCEDURE_CALL
        else
            ASSIGNMENT;
```

Very few problems were encountered in compiling SUILVEN statements. Those difficulties which arose were caused by the fact that the B1700 micro-architecture is unsympathetic to the needs of a high-level language, principally because of the lack of provision for the storage of intermediate results in expression evaluation.

Adapting SUILVEN to the B1700 micro-architecture has resulted in inefficiencies in the machine code generated for certain high-level constructs, notably:-

(i) Function designators used in expressions.

(ii) Conditional expressions.

(iii) Subscripted variables.

All these constructs require the use of some intermediate storage area. In fact, extra code is generated to move these intermediate results to and from the B1700 address stack which is used for temporary storage of results. Inefficiencies and optimisation will be discussed in more detail in section 5.7.

5.6 SUILVEN Input/Output Features

The reader will have observed from the previous chapter that SUILVEN input and output statements are not defined as part of the language. As the I/O requirements of the s-machines which we have implemented have been minimal, only simple standard procedures have been included in SUILVEN to provide an I/O facility.

The SUILVEN I/O procedures require the programmer to use two I/O buffers called INPUT_BUFFER and OUTPUT_BUFFER. Associated with each of these buffers is a pointer respectively named INPOINT and OUTPOINT. These variables are automatically predeclared in all SUILVEN programs and may be accessed like all other variables declared in a SUILVEN program.

There are five I/O functions which operate either on INPUT_BUFFER or OUTPUT_BUFFER.

(i) READ

Reads a card image from the input stream into INPUT_BUFFER.

(ii) WRITE

Writes OUTPUT_BUFFER to the line printer in character form.

(iii) PUT(<expression>)

Evaluates the given expression, converts the result to characters and moves these characters to OUTPUT_BUFFER.

(iv) PUTSTRING(<string>)

Moves the specified string to OUTPUT_BUFFER.

(v) CLEAR

Sets all characters in OUTPUT_BUFFER to blank.

These simple I/O statements have been adequate for the testing and evaluation of those s-machines which have been implemented in SUILVEN.

The use of I/O statements for s-machine evaluation significantly degrades s-machine performance. This is due to the fact that characters may only be moved singly to OUTPUT_BUFFER. The reader will realise the effect of this on machine performance when diagnostic strings are output.

Naturally, should a language like SUILVEN be used in a production rather than a research environment, the language input/output facilities would have to be improved. We envisage that this is best achieved using a descriptor based system but, considering the present usage of SUILVEN, we feel the standard procedure I/O system is adequate.

5.7 Code Optimisation

The traditional requirement of microprograms is that they be as efficient as possible. All sorts of programming 'tricks' are used to increase the speed and reduce the size of microprograms.

We believe that this view is destined to change, as it is now changing with regard to high-level languages. As technological developments increase the speed and decrease the cost of fast access store, microprograms will have a wider application, and reliability and easy maintenance rather than efficiency will be the prime concern of the microprogrammer.

At present, however, if a high-level microprogramming language is to be at all credible, the generated microcode must be comparable in efficiency with hand-written code. We have, therefore, given a good deal of thought to optimising the code output by the SUILVEN compiler, although no automatic optimisation techniques have yet been implemented.

This section of the thesis discusses some of the problems of optimising B1700 microcode and the techniques we have used to optimise the code. These techniques are all reliant on programmer intervention by the s-machine implementor.

5.7.1 Microcode Inefficiencies

The generation of optimal microcode from a high-level language program is hampered by a number of factors:-

(i) Micro-architecture is not designed to support high-level languages. Consequently, problems arise in generating efficient code for some high-level constructs such as arithmetic expressions

(ii) Microprogrammable machines usually have a number of general purpose registers. There is a very significant difference in execution speed between register-register and register-store instructions. Typically, register-register operations are 5 times faster than register-store operations and, as a result, it is obviously beneficial to optimise the use of registers. If attempted automatically, this is a non-trivial problem.

(iii) There is usually no multiply microinstruction available on microprogrammable machines. The implementation of arrays where the user may define his word size in bits and access the array as an array of words requires a multiplication. For example, if the array A was composed of 32-bit words, the bit address of the nth element is computed:-

$$BASE(A) + 32 * (n-1)$$

This essential multiplication introduces very significant microprogram overhead as it must be executed using a sequence of microinstructions.

The solution to these problems, in the long run, requires the modifications of micro-architectures so that they support high-level languages more efficiently. In our implementation of SUILVEN, the above problems have resulted in some serious inefficiencies in the microcode generated by the compiler. Hence, some attempts have been made to eliminate redundant microinstructions and to minimise the number of register-store data transfers.

5.7.2 Minimising Register-Store Data Transfers

The B1700 has a group of fast storage registers known as the scratchpad registers. Normally, variables declared in a SUILVEN program are allocated addresses in the machine store, but it is obviously desirable to utilise the scratchpad registers for the storage of frequently accessed variables.

As the compiler cannot know in advance which variables will be most used, compiler directives have been introduced which allow the user to specify which program variables are to be stored in the scratchpad. We assume that the s-machine programmer has some intuitive notion of how frequently each variable in his program is accessed. The compiler directives are:-

(i) SCRATCHPAD <list of variable names>

This specifies that the named variables should be allocated to scratchpad registers.

(ii) COPYSCRATCH <list of variable names>

This causes code to be generated which will copy the specified variables from store into a scratchpad register.

(iii) CLEARSCRATCH <list of variable names>

This causes code to be generated which will copy the specified variables from the scratchpad to store, thus freeing their scratchpad locations for subsequent use.

As all variables always have a location reserved for them in store it is permissable to transfer a variable to the scratchpad on entry to a routine which uses that variable frequently. It may be returned to store on exit from that routine.

The performance improvement obtained by retaining variables in the scratchpad rather than in store is clearly demonstrated by the example below.

Example

This example compares the generated microinstructions from a SUILVEN assignment statement when the operands are in store, and when they are in the scratchpad.

Consider the assignment:-

A := A + B

The microinstructions generated when both A and B are in store are shown overleaf.

Microinstruction	Number of machine cycles
FA:=Address(A)	2
X:=READ	5
FA:=Address(B)	2
Y:=READ	5
X:=SUM	1
FA:=Address(A)	2
WRITE(X)	6

Total number of microinstructions generated = 7

Total number of machine cycles used = 23

This may be compared with the microinstructions generated for the same assignment statement when both A and B have been allocated scratchpad locations.

Microinstruction	Number of machine cycles
X:=Scratchpad(A)	1
Y:=Scratchpad(B)	1
X:=SUM	1
Scratchpad(A):=X	1

Total number of microinstructions = 4

Total number of machine cycles used = 4

This represents a reduction of about 50% in the number of microinstructions but, because of the elimination of store-register data transfers, an execution time improvement of about 500% is achieved.

The reader will recall that a field named SPAD exists in the symbol table entry for each declared name. This field is used to hold the scratchpad location of that variable, should it have been assigned by the programmer.

The ability to retain variables in the scratchpad can also significantly reduce the number of array accesses made when a stack is implemented in SUILVEN. It is obviously impossible to retain all stack elements in fast scratchpad registers, but the top few stack elements may be kept in the scratchpad rather than in store. If this is done, very few store accesses need be made when the stack is used for computation.

We have carried out a number of simulations to determine the optimum number of registers needed as top stack registers. Our results are tabulated below:-

Number of stack registers	Average % of store transfers
0	100
2	11
4	3
6	2

TABLE 5.1

STACK REGISTERS AND PERCENTAGE OF STORE ACCESSES

Naturally, this method of stack implementation makes for more complex stack manipulation routines. The more elements retained in registers, the more complicated become the stack push and pop procedures. In the s-machines which we have implemented, two stack registers have been used. We estimate that the overhead involved in the more complex stack routines does not justify the decrease in store accesses which would result from using more stack top registers. Typically, the execution speed of a stack machine is increased by about 30% by using two stack top registers.

5.7.3 Eliminating Redundant Microinstructions

This section of the thesis describes how redundant microinstructions are necessarily generated by the SUILVEN compiler, how they may be eliminated, and the improvement possible by so doing. No automatic techniques have been developed to remove these instructions but aids have been provided to assist hand optimisation.

Redundant microinstructions occur most commonly in two constructs translated by the SUILVEN compiler:-

(i) Conditional expressions

(ii) Operations on structured data areas

In the former case, the redundancy is a direct result of the generality of SUILVEN conditional expressions. Recall that such an expression has the form:-

<expression> <relation> <expression>

Expressions are always evaluated into a B1700 register called the X register and comparisons are made using this register and the Y register. Therefore, the code generation sequence for a conditional expression is:-

```
X := <left side>
Save X on address stack
X := <right side>
Y := top of stack
Compare
```

If, as is most common, each side of the conditional expression is a simple variable, this code sequence may be reduced to:-

```
X := <left side>
Y := <right side>
Compare
```

Two redundant microinstructions may be eliminated.

When working with structured data areas, it is common to manipulate more than one field of a particular structure. The SUILVEN compiler treats each field separately and generates code to compute the field address in each case.

However, the B1700 has an automatic address updating feature which increments or decrements the store address register in parallel with the store access operation. When judiciously used, this can make a significant contribution to microcode efficiency. This is illustrated by the example overleaf:-

Example

The variables A is structured into two fields called HEAD and TAIL. Consider the following assignment statements:-

```
A.TAIL := B;
A.HEAD := C;
```

Assuming B and C are held in the scratchpad and A in store, the code generated by the SUILVEN compiler is:-

```
X := Scratchpad(B)
FA := Address(A.TAIL)
WRITE(X)
X := Scratchpad(C)
FA := Address(A.HEAD)
WRITE(X)
```

If the automatic address updating feature in the B1700 is used, the second assignment to FA may be eliminated:-

```
X := Scratchpad(B)
FA := Address(A.TAIL)
WRITE(X) DEC FA
X := Scratchpad(C)
WRITE(X)
```

Obviously, this is most beneficial when operating on structured array elements, as the need to recompute the array index is eliminated. This reduces the number of microinstructions and significantly increases program execution speed.

It is very difficult for the SUILVEN compiler to generate code which uses the address updating feature of the B1700. The code to evaluate an expression operand is generated before consuming the next operand from the input stream. Hence, there is no way of knowing whether operands are adjacent in store and of using the address updater accordingly.

To remove the redundancies in conditional expressions and to utilise address updating, the machine code generated by the SUILVEN compiler may be hand optimised.

This approach is not usually adopted with high-level languages as the code generated by the compiler is not usually geared towards the human reader. An additional problem with hand optimisation is that the displacement in branch instructions must be modified when instructions are deleted. This is a tedious and error prone task. We have developed facilities to simplify the hand optimisation of the generated code. These are:-

(i) The microcode generated by the SUILVEN compiler for each language statement may be listed adjacent to the statement. This code listing is in an easily readable mnemonic form.

(ii) A special purpose editor has been written which enables the generated microcode to be modified. This editor takes care of changes in branch displacements when instructions are added or deleted. Clearly, the consistent control structure of SUILVEN, and the lack of undisciplined branching in a SUILVEN program simplifies the construction of such an editing program.

The hand optimisation of an s-machine whose size is about 2000 microinstructions can be accomplished fairly quickly. We estimate that approximately 95% of redundant microinstructions can be eliminated if two optimising passes are made through the object code.

To give some indication of the effectiveness of hand optimisation, the histograms below show the number of microinstructions in an s-machine before and after optimisation. The programs used to gather this information were:-

(i) An s-machine for a lamda-calculus language(SASL)

(ii) A general purpose stack machine for PASCAL

(iii) A simple stack machine(SIMPS), developed initially as a vehicle to test the SUILVEN compiler.

Both (i) and (ii) above are described in detail in chapter 6.

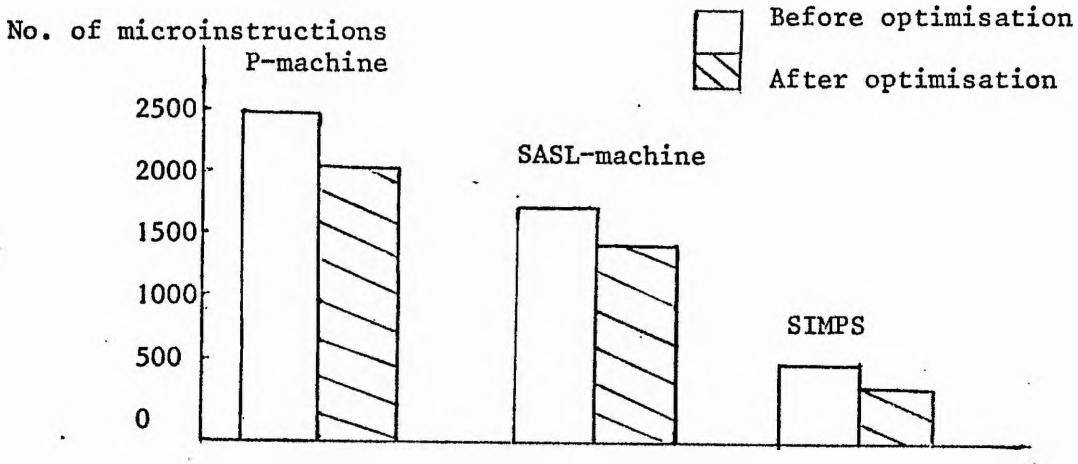


FIGURE 5.6

SIZE COMPARISON OF OPTIMISED AND UNOPTIMISED S-MACHINES

The approximate reduction in the number of microinstructions for each s-machine is as follows:-

SASL machine	20%
PASCAL machine	15%
SIMPS machine	14%

The discrepancy between the figures for the SASL machine and the figures for the other s-machines may be accounted for. The SASL machine is a higher level machine than the comparitively simple stack machines. As a result, the program emulating that machine contains relatively more conditional expressions and operations on structured data areas. As microinstruction redundancy is most obvious in these contructs, their optimisation results in a relatively greater reduction in the size of the SASL machine.

5.8 The Lineprinter Output Produced by the SUILVEN Compiler

The format of the lineprinter output from a compiler is often a feature which is neglected by the compiler writer. However, as this output is the sole means of communication between the compiler and the programmer, we believe that much care should be taken over its design.

A good listing contains much more than a printout of the source text. It must establish a co-ordinate system within the program by which program elements may be identified, both for human communication and for association with compiler error messages.

Accordingly, the listing produced by the SUILVEN compiler is distinguished by the following features:-

(i) An informative heading providing information about the compiler and compiler options.

(ii) SUILVEN statement numbers and card sequence numbers.

(iii) If a statement is within a procedure, the procedure name is printed by the statement.

As the full width of the lineprinter carriage is used to provide this information, it is not possible to give examples of the compiler output here and still retain neatness. However, Appendix 3 consists of SUILVEN program listings where the output format may be examined.

5.9 Statistics concerning the SUILVEN Compiler

The evaluation of a compiler is only meaningful when it is compared with similiar compilers. For example, Witchman(W8) has compared a number of ALGOL compilers, and Wortman(W9) a number of PL/1 compilers. As no other SUILVEN compiler exists, it is impossible to evaluate the compiler objectively. However, some figures concerning details of the compiler implementation are given below.

Approximate Compilation Rate	:	300 SUILVEN statements per minute
Store Occupied by the Compiler	:	75K bytes
Store Occupied by the Compiler plus the SPITBOL system	:	140K bytes
Compiler Development Time	:	9 months

Unquestionably, the compilation rate could be increased and the store requirements decreased had a language other than SNOBOL4 been used to implement the compiler. However, this would probably have resulted in an increase in compiler development time. As SUILVEN is a research tool, rapid compiler development took priority over the construction of a fast/compact system.

5.10 Summary

In this chapter, the development of a compiler for the high-level microprogramming language SUILVEN has been described. Emphasis has been placed on discussion of design decisions, compilation of SUILVEN's data description features, and the optimisation of the microcode generated by the compiler. Because the techniques are well known, the compilation of SUILVEN statements is only discussed briefly.

We believe that the construction of the SUILVEN compiler has been a successful part of our research project. Improvements could be made in the compiler error recovery strategy and, possibly, some automatic code optimisation could be attempted. However, as SUILVEN is a research tool, designed for a limited purpose, it is arguable whether the cost of implementing such improvements is justified.

CHAPTER 6

THE IMPLEMENTATION OF ABSTRACT MACHINES

In this chapter, the implementation of interpreters for two non-trivial abstract machines is discussed. These abstract machines are:-

(i) A stack machine used to implement PASCAL.

(ii) A list-oriented machine used to implement SASL, a locally developed lambda-calculus programming language.

SUILVEN implementations of these machines have been programmed by the author of this thesis. These implementations are compared with other implementations in both high-level and low-level programming languages. In particular:-

(i) With the PASCAL stack machine programmed in PASCAL and in PL360.

(ii) With the SASL machine programmed in BCPL and MIL.

This chapter, therefore, begins with some general discussion concerning the characteristics of s-machine interpreters. This is followed by a description of the PASCAL s-machine. The various implementations of that machine are then compared paying particular attention to the descriptive power of the languages used to implement the interpreters.

The final sections of the chapter provide a corresponding description of the SASL machine and its implementation. The chapter concludes with a general evaluation of SUILVEN as a machine description and implementation language.

The implementation of the PASCAL and SASL s-machines occupied some months. As will become clear later, these machines have a radically different structure, with the PASCAL machine being a fairly simple stack machine of a conventional design. The SASL machine, on the other hand, is a true high-level machine which utilises high-semantic content instructions, a list-oriented store, and a tagged data architecture.

SUILVEN was therefore tested using significantly different s-machines. These machine implementations provided insights into both the good and the bad features of the language design.

6.1 Abstract Machine Interpreters

The structure of abstract machine interpreters mimics the instruction fetch-decode-execute sequence on 'hard' machines.

The main interpreter loop checks for interrupts, handles them if necessary, fetches and identifies a machine instruction, and executes the appropriate sequence of operations to interpret that instruction. A skeleton of a typical interpreter is shown overleaf in figure 6.1.

```
INTERPRETER
        INITIALISE
        while interpreting do
            {
                if any_interrupt then
                    HANDLE_INTERRUPT
                FETCH_INSTRUCTION
                case instruction op code of
                                :
                    Code to execute each machine instruction
                                :
                endcase
                   else
                      ERROR
            }
END INTERPRETER
```

FIGURE 6.1

THE SKELETON STRUCTURE OF AN S-MACHINE INTERPRETER

This structure, with a greater or lesser degree of enhancement depending on the machine being implemented, is fundamental to all s-machine interpreters. Clearly, SUILVEN has the necessary control facilities to implement such a structure.

6.2 The PASCAL Machine

The architectural features of an abstract machine for PASCAL(W1) are described in this section. We assume that the reader has some knowledge of the general features of that language. The machine was designed by Jensen(J1) as part of a project to provide a portable implementation of PASCAL.

As PASCAL is a block-structured language, it is natural that the PASCAL machine(subsequently referred to as the P-machine) should be a stack machine. To support PASCAL's record structures, a heap is also an integeral part of the machine architecture. The diagram below illustrates the machine organisation:-

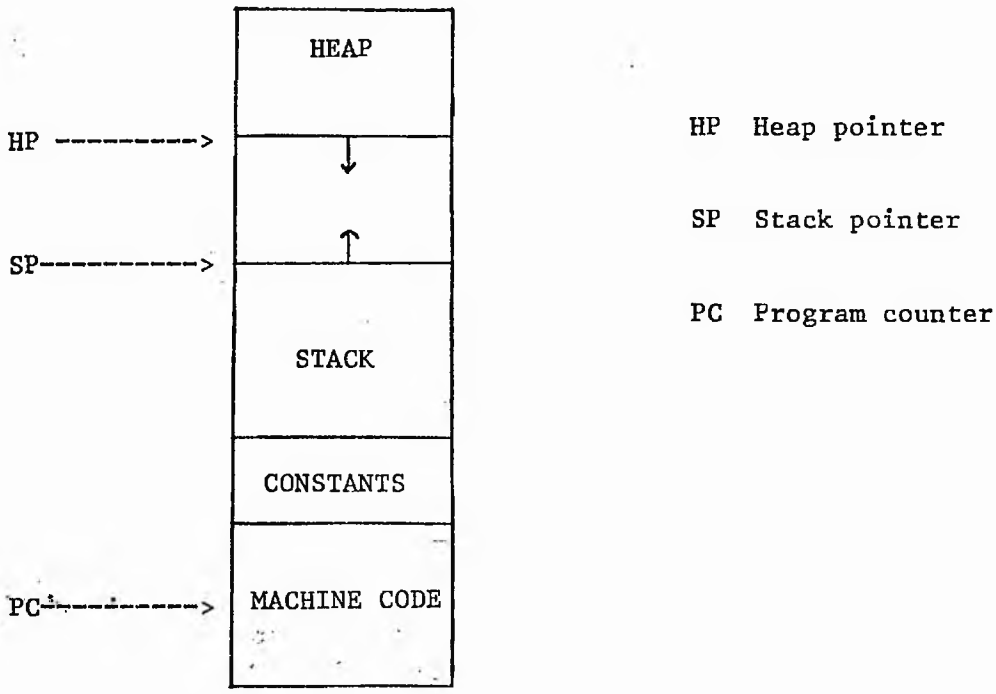


FIGURE 6.2

THE ORGANISATION OF THE PASCAL S-MACHINE

It should be noted that the heap is not a true heap but is organised on a LIFO basis. The machine has no mechanism for garbage collection.

A P-machine instruction is split into three fields:-

(i) The op code

(ii) The P field

(iii) The Q field

The P and Q fields are not used in all instructions but, for convenience, a consistent instruction format has been used. The instruction set consists mostly of instructions such as ADD, MULTIPLY, EQ, GRT, etc which operate on the stack. In addition, there are a number of instructions which have an address as a parameter. These are used for loading data onto the stack and storing information from the stack. There are some instructions which are specifically oriented towards features of PASCAL. These instructions include operations such as:-

(i)	INN	Tests for set membership
(ii)	INT	Set intersection operator
(iii)	ODD	Tests if the top of the stack is odd

Naturally, instructions for array bound checking, procedure entry and procedure exit are included in the machine instruction set.

The P-machine can therefore be summarised as a typical stack machine, not dissimiliar to Randell and Russell's Beta machine(R1). The machine has been deliberately organised in this simple manner so that it may be more portable.

6.3 Implementing the P-machine in SUILVEN

The SUILVEN implementation of the P-machine is described here with a listing of the program displayed in appendix 3.

The organisation of the data areas in the P-machine is exceptionally simple and is easily described using two SUILVEN BITS declarations:-

```
BITS(WORD_SIZE)  ARRAY(STORE_SIZE)   STORE;

BITS(WORD_SIZE)  STACKPOINTER,HEAPPOINTER,PROGRAMCOUNTER:
```

A number of other variables are declared which act as internal machine registers. These include variables to hold a machine instruction, an op code register and an interrupt register.

The implementation of many of the stack machine instructions is correspondingly simple. Most instructions are executed by either one or two SUILVEN statements. For example:-

```
Add Integer    :    PUSH(POP() + POP())

Load Address   :    PUSH(BASE(P) + Q)

Branch false   :    IF POP() = 0 THEN
                         PROGRAMCOUNTER := Q
```

An interesting point which emerged from the implementation of the P-machine was that there was little need for local BITS variables. Those which were used, could easily have been replaced by global variables, without causing any problems or confusing the program structure.

However, use was made of locally declared templates in routines which interpreted real and integer arithmetic operations. For example, the template for a real number is declared:-

```
TEMPLATE  REAL = CHAR(7), SIGN(1), MANTISSA(16)
```

whereas that for an integer is:-

```
TEMPLATE  INTEGER = SIGN(1), NUMB(23)
```

The positioning of the sign bit to the right rather than to the left of the mantissa in a real number is forced on the s-machine implementor by the operation of the B1700 arithmetic unit.

SUILVEN was found to be an adequate language for implementing the PASCAL machine. Apart from real operations, which often present implementation problems, SUILVEN easily implemented and described the data areas and operations of the abstract machine.

6.3.1 Implementation Data for the SUILVEN P-machine

Table 6.1 below summarises miscellaneous items of quantitative information concerning the SUILVEN implementation of the P-machine.

Number of SUILVEN statements	570
Number of microinstructions generated by the SUILVEN compiler	2342
Number of microinstructions after hand optimisation	2010
Implementation time	6 weeks

TABLE 6.1

P-MACHINE INTERPRETER INFORMATION

The number of microinstructions after hand optimisation is 2010 and this figure represents 32160 bits of information. This may be compared with the 28K-32K bits figure specified by Wilner(W3) as the size of a typical s-machine programmed in MIL. Such a machine, although slightly more complicated than the P-machine, is of the same order of complexity.

Therefore, it appears that encoding a fairly low-level s-machine in SUILVEN involves only a slight overhead in terms of numbers of microinstructions compared to a hand coded machine. We suspect

however, that a hand coded machine operates more efficiently because better use is made of fast registers and store accesses are minimised.

Notice that figures are not given for the execution speed of the P-machine. It has not been possible to collect comparitive information concerning the machine speed as an MIL version of the machine is not available to us.

6.4 A Comparison of P-machine Implementations

In this section, the SUILVEN implementation of the P-machine is compared with other P-machine interpreters encoded in PASCAL and PL360. Naturally, as microcode is not generated by the PASCAL and PL360 compilers, it is not possible to give an objective comparison of the interpreters. Rather comparisons are made on three bases:-

(i) The number of statements in each interpreter

(ii) The readability of each interpreter listing

(iii) The suitablity of each language as an MDL

6.4.1 SUILVEN and PASCAL

The P-machine interpreter used for this comparison is the definitive version of the interpreter supplied by the P-machine designers. The designers claim(J1) that the interpreter constitutes an adequate description of the P-machine and provide only minimal extra documentation about the machine.

When comparing this program with the SUILVEN version of the interpreter, the reader must bear in mind that some operations, such as set operations and real arithmetic operations, are implemented in PASCAL using themselves. For example, the set operation INN is encoded:-

```
SP := SP - 1;
STORE(.SP.).VB := STORE(.SP.).VI  IN  STORE(.SP+1.).VS;
STORE(.SP.).STYPE := BOOL;
```

While this is a simple way of implementing the instruction, it has the disadvantage, as far as machine description is concerned, of giving no indication of how sets and set operations are implemented as bitstrings.

In spite of PASCAL's advantage in this respect, the sizes of the PASCAL and SUILVEN P-machine interpreters are roughly comparable.

This may be explained by the fact that PASCAL has a very strict type discipline which is not particularly appropriate in the implementation of interpreters. The type discipline necessitates that the P-machine stack is implemented as a tagged structure, with the tag indicating the type of operand on the stack. This is illustrated in the example above. Naturally, extra code is needed to accomodate this tagging and this increases the overall size of the interpreter.

As would be expected, the implementation of simple machine instructions such as ADI(Add Integer) and UJP(Unconditional Jump) is similiar in PASCAL and in SUILVEN. For example:-

Add Integer(ADI)

PASCAL

```
SP := SP - 1;
STORE(.SP.).VI := STORE(.SP.).VI + STORE(.SP+1).VI;
```

SUILVEN

```
PUSH(POP() + POP())
```

Because of the particular implementation of the P-machine stack, using stack top registers, the SUILVEN add instruction is implemented as procedure calls to push and pop routines rather than as a direct stack manipulation.

In terms of readability, both the PASCAL and SUILVEN P-machine interpreters are reasonably clear and easy to understand. Readability is significantly affected by programming style, and it is our opinion that the style displayed in the PASCAL interpreter program leaves a great deal to be desired. In particular, we deplore the use of one and two character identifiers and the lack of comments in the program.

Disregarding this however, we believe that the SUILVEN interpreter is slightly easier to understand than the PASCAL interpreter. This is due to the fact that a reader not completely familiar with PASCAL may find the tagged stack architecture rather confusing, with the rigid type discipline concealing rather than displaying the salient features of the P-machine.

Our general opinion of the merits of SUILVEN and PASCAL as machine description languages is that they are comparable. While SUILVEN's data description features allow the machine data areas to be more exactly specified, PASCAL's higher level operators can be used to provide an extra level of abstraction.

6.4.2 SUILVEN and PL360

This section consists of a brief comparison of P-machine interpreters encoded in SUILVEN and PL360. The PL360 version was locally programmed to bootstrap PASCAL onto an IBM S/360 computer.

PL360, designed by Wirth(W7), is a machine oriented programming language for the IBM S/360-370 range of computers. It superimposes high-level language features such as procedures, control statements, and expressions onto the S/360 assembly code. These high-level features make PL360 much easier to use and understand than assembly code.

Nevertheless, PL360 still remains a low-level language as is reflected in the comparison of the sizes of the SUILVEN and PL360 P-machine interpreters. In spite of the availability of real arithmetic features(the real arithmetic package is a large part of the SUILVEN interpreter), the PL360 interpreter has about 5 times as many statements as the SUILVEN interpreter.

Because of PL360's excellent control structures, the interpreter description is surprisingly clear. The well structured nature of the interpreter means that machine language is introduced at a fairly low level. However, the SUILVEN version of the interpreter

provides a more readable and understandable description of the P-machine. This is to be expected as SUILVEN as a much higher level language than PL360.

SUILVEN's power as an interpreter programming language is illustrated by the time taken to encode each interpreter. Encoding the P-machine in SUILVEN occupied about 6 weeks, whereas programming the interpreter in PL360 took about 5 months.

6.5 The SASL Machine

This part of the thesis describes the design of an abstract machine for a list processing language called SASL. This machine, based on Landin's SECD machine(L2), is defined and described by Turner(T2) and Nelson(N1). As SASL is not widely known, a very brief overview of the language is given below.

SASL is a descriptive language, defined by Turner(T1). Fundamentally, it is a convenient notation for expressing the lambda calculus. The language has no goto statement, no assignment statement, and no explicit iteration features. Loops are programmed using recursion. A SASL program is an expression and its outcome is to print the value of that expression.

SASL is intended for list processing and recognises four basic types of object - integers, characters, truthvalues, and lists. Types are not checked at compile time but at run time by the SASL machine interpreter. As the language is based on the lambda calculus, functions may be treated as any other object and may be arguments to other functions and function results.

Even from the very brief description above, it is obvious that SASL is unlike most conventional programming languages such as PASCAL. Their s-machines, correspondingly, have completely different architectures.

The SASL machine is based around a list area and a stack. The s-machine code and data are held in the list area, with a register called the C register pointing to the next machine instruction to be executed. Another register, called the E register, is the environment register. That is, the E register points to the list of data items accessable to the program at any one time. This organisation is diagrammed below in figure 6.4.

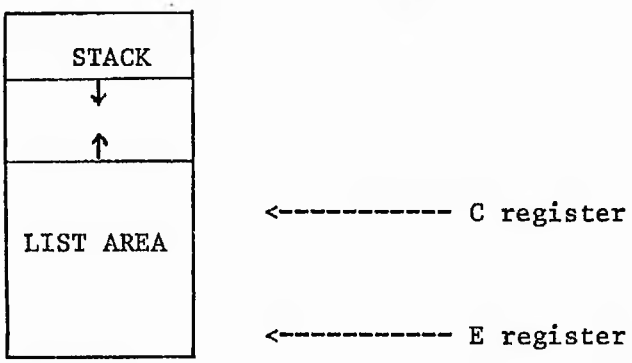


FIGURE 6.4

THE ORGANISATION OF THE SASL MACHINE

The elements in the list constitute both machine instructions and data. Each element is structured into four fields:-

(i) A tag bit, used in garbage collection

(ii) A field specifying the type of the element

(iii) A field holding the value of the list element

(iv) A pointer to the following element

The stack is more simply structured into a type field and a value field.

The instruction set of the SASL machine is made up of two kinds of instructions:-

(i) Fairly simple stack instructions appropriate for a list processing environment. These instructions include ADD, EQ, GEQ, HEAD, TAIL, etc, etc.

(ii) Instructions which have one parameter and which generally perform more complex operations than the simple stack instructions.

These latter instructions are more interesting than the simple stack instructions. They include instructions to update the program environment list, instructions to handle recursive declarations, and instructions to manipulate functions. The combination of these instructions with the stack operations results in a simple, elegant, and powerful abstract machine.

6.6 Implementing the SASL Machine in SUILVEN

This section deals with the SUILVEN implementation of the SASL machine. This was a more complex and difficult program to construct than the P-machine interpreter. A complete listing of the SUILVEN version of the SASL machine interpreter is available in appendix 3.

The main data areas of the SASL machine are defined using BITS declarations:-

```
BITS(STACK_WIDTH)  ARRAY (STACK_SIZE)  STACK;

BITS(LIST_WIDTH)  ARRAY (LIST_SIZE)  LIST;

BITS(WORD_SIZE)  CREG , EREG;
```

The stack and list areas are structured using the following template declarations:-

```
TEMPLATE  STACKSTRUCTURE = TAG(TAG_SIZE), DATA(WORD_SIZE);
TEMPLATE  LISTSTRUCTURE = GC(1), TAG(TAG_SIZE), HEAD(WORD_SIZE),
                                  TAIL(WORD_SIZE);
```

In order to minimise the number of store accesses, scratchpad registers are used to hold the top stack elements with the stack push and the stack pop routines programmed accordingly.

The overall structure of the stack machine is that described in section 6.1. That is, the program consists of a while loop fetching an instruction, checking for interrupts, then executing that instruction. The SASL machine interpretation loop is a little

more complex than the simple loop in section 6.1, as it includes provision for run-time type checking of instruction operands.

The implementation of many of the SASL machine instructions in SUILVEN is fairly straightforward. Like the PASCAL machine, most instructions can be interpreted by a few SUILVEN statements. In the examples below, it may be assumed that run-time type checking has been carried out when the instruction is fetched from the list area.

Instruction HEAD

> " *The top of the stack is a pointer to a list. Replace this pointer with the element at the head of the list. The top of stack data element is held in a register called TOPS* "

```
PUSH(LIST(.TOPS.).TAG, LIST(.TOPS.).HEAD);
```

Notice that the stack push procedure takes two parameters - the tag holding the data type and the data value.

Instruction ADD

> " *Adds the top stack elements* "

```
PUSH(INTEGER_TYPE, TOPS + SECS);
```

Instruction COMMA

" *This instruction appends an item onto a list. The item to be appended is in the second top stack element and a pointer to the list is on top of the stack. A pointer to the new list is pushed onto the stack* "

```
TR := GET_A_NEW_LIST_CELL();
LIST(.TR.).TAIL := TOPS;
LIST(.TR.).TAG := POINTER_TYPE;
LIST(.TR.).HEAD := SECS;
PUSHSTACK(POINTER_TYPE, TR);
```

The implementation of some of the more complex SASL machine instructions, notably those handling declarations, was more difficult. The instructions which deal with SASL declarations are as follows:-

(i) DECL <name>

Removes the top item from the stack and makes a new entry in the environment list, associating <name> with the value taken from the stack.

(ii) DECLGUESS <name>

Makes a new entry in the environment list associating <name> with the unknown value GUESS. This is used when dealing with recursive declarations where the item to be associated with <name> is undefined.

(iii) TIEKNOT <name>

The environment contains <name> associated with the value "GUESS". This instruction pops the stack and overwrites GUESS with the value popped from the stack.

As <name> in each of these instructions may be the name of a list type entity, which may itself be composed of lists, the reader will appreciate that the most natural implementation of these instructions uses recursion. Unfortunately, such a facility is not inherent in SUILVEN, and the programmer must use the SASL machine stack to save and restore local variables when recursive routines are to be implemented.

All three of the above instructions are concerned with modifying the environment list in some way. In order to minimise the number of recursive procedures, the approach which we have adopted in implementing these instructions is to flatten all lists, thus ensuring that the instructions may be interpreted in an iterative manner. List flattening means that each list component which is itself a list is replaced by the actual elements of that list, resulting in a final list which is strictly linear.

The instructions DECL, DECLGUESS, and TIEKNOT all call the recursive procedure FLATTEN to accomplish this list flattening. The examples overleaf illustrate this procedure and the procedure DECL which implements the DECL and DECLGUESS instructions.

```
PROCEDURE FLATTEN(BITS(WORD_SIZE) L1,LAST);
```

" The effect of this procedure is to flatten the list pointed at by L1, i.e. all list elements which are lists are replaced by the actual list components. FLATTEN may be called recursively, and if so, LAST points to the element immediately preceding the element whose value is L1. LAST is used when replacing the list descriptor by the list components. The recursion level is held in the global variable RECLEVEL "

```
IF LIST(.L1.).TAG = POINTER_TYPE THEN
    £(
    " List element is a list - chain it onto previous list
      pointed to by LAST. "
    PUSHSTACK(POINTER_TYPE,L1);
    L1:=LIST(.L1.).HEAD;
    LIST(.LAST.).TAIL:=L1;
    RECLEVEL:=RECLEVEL+1;
    £)
ELSE
    IF LIST(.L1.) = NIL THEN
        £(
        " If processing subsidiary list chain its last element
        onto the next elemnt in higher level list. Restore P
        from the stack and reduce RECLEVEL. If RECLEVEL is
        0 we are at end of main list so exit "
        IF RECLEVEL > 0 then
            £(
```

```
            RECLEVEL:=RECLEVEL-1;
            FILL_TOPSTACK_REGISTERS();
            LAST:=L1;       UNSET(TOPSF);
            L1:=LIST(.TOPS_DATA.).TAIL;
            LIST(.LAST.).TAIL:=L1;
            £)
        ELSE
            " End of highest level list so exit   "
            EXIT
        £)
    ELSE
        " Element is not a list so no flattening required.
        Move on to next element   "
        £(
        LAST:=L1;     L1:=LIST(.L1.).TAIL;
        £);
    " Now call FLATTEN recursively. Unless the pointer
    L1 has been stacked this is not really a recursive call
    but is merely a jump back to the beginning of the procedure
    to repeat the operation sequence on the next list element  "
    FLATTEN(L1,LAST);
END "FLATTEN" ;
```

```
PROCEDURE DECL();

    " This procedure processes SASL declarations.  If the declaration
    is recursive, the value to be associated with the given name is
    not known when the declaration is processed so the unknown value
    GUESS is filled in.  A DECLGUESS operation is identified by the
    setting of a global flag called GUESS_FLAG.  The instruction
    parameter points to the list of names to be added to the
    environment and , if a DECL instruction, the top of the stack
    points to the associated value list  "

    BITS(WORD_SIZE) OP;
    OP:=NIL;
    PUSHSTACK(GET_PARAMETER());
    FLATTEN(TOPS_DATA,OP);
    IF UNSET(GUESS_FLAG) THEN
        FLATTEN(SECS_DATA,OP);
    REPEAT
        £(
        ADD_NAME_AND_VALUE_TO_ENVIRONMENT();
        TOPS_DATA:=LIST(.TOPS_DATA.).TAIL;
        IF UNSET(GUESS_FLAG) THEN
            SECS_DATA:=LIST(.SECS_DATA.).TAIL;
    UNTIL
        TOPS_DATA = NIL
    DO  ;

END "DECL" ;
```

6.6.1 Data on the SASL Machine Implementation

The table below summarises miscellaneous items of quantitative information about the SUILVEN program which interprets the SASL machine.

Number of SUILVEN statements	520
Number of microinstructions generated by the SUILVEN compiler	1895
Number of microinstructions after hand optimisation	1480
Implementation time	10 weeks

TABLE 6.2

SASL-MACHINE INTERPRETER INFORMATION

Notice that the decrease in the number of microinstructions after optimisation is greater than that achieved with the P-machine. This is a direct result of the more frequent use of operations on structured data elements in the SASL machine. The code generated by the compiler for sequences of these operations contains redundant microinstructions and removal of these results in a significant decrease in the size of the s-machine.

Notice also that the size of the SASL s-machine after optimisation is about 24k bits. This is significantly smaller than the average

s-machine size quoted by Wilner(30K bits). To account for this decrease in size, we can only conjecture that the design of most s-machines contains inherent redundancies, such as different load instructions for loading different types of data. The generality and elegance of the SASL machine design appears to have eliminated these redundancies resulting in a more compact interpreter.

6.7 A Comparison of SASL-Machine Implementations

As well as the SUILVEN implementation of the SASL-machine, implementations exist which have been programmed in BCPL and MIL. In this section, the SUILVEN version of the SASL-machine is compared with these other interpreters.

SUILVEN and BCPL are compared as s-machine programming languages using the same bases of comparison as were used with PASCAL:-

(i) The number of statements in each version of the interpreter.

(ii) The readability of the interpreter.

(iii) The suitability of each language as a machine description language.

However, as MIL is the B1700 micro-assembler, the SUILVEN version and the MIL version of the SASL-machine are not compared as machine descriptions. Instead, a comparison is made between the microcode generated by the SUILVEN compiler and the MIL program.

6.7.1 SUILVEN and BCPL

The programming languages BCPL and SUILVEN display some similiarity inasmuch as neither language requires the types of variables to be declared. Hence, the awkwardness of PASCAL's strict type discipline is not a feature of BCPL programs used to implement interpreters.

As a result, the SUILVEN and BCPL code which interprets the simpler stack operations of the SASL machine is very similiar. For example:-

Instruction ADD

SUILVEN

```
CHECK_TYPES_ON_STACK(INTEGER_TYPE);
PUSH(INTEGER_TYPE,TOPS+SECS);
```

BCPL

```
A := NUMBER(A);
B := NUMBER(B);
PUSHN(A+B);
```

The BCPL function NUMBER checks that the type of its parameter is a number and the procedure PUSHN pushes a number onto the stack.

However, in spite of the similiarity of the implementation of the simpler SASL-machine instructions, the BCPL interpreter contains around 350 statements - significantly fewer than the SUILVEN interpreter which is about 520 statements long.

This difference is attributable to two factors:-

(i) More use of procedures for code sharing is made in the BCPL program. The SUILVEN interpreter uses similiar in-line code in many places and this could be replaced by a procedure call. This programming style of using in-line code rather than procedures was adopted to avoid the inevitable overhead associated with calling and returning from a procedure.

(ii) BCPL has a very wide variety of control constructs, including recursion. As a result, the coding of the more complex SASL-machine instructions is more concise in BCPL than in SUILVEN.

In spite of the fact that the BCPL version of the interpreter is more concise, we consider that it is less readable and understandable than the equivalent SUILVEN program. In short, the SUILVEN program is a better description of the SASL machine.

This difference in readability is attributable to two factors:-

(i) BCPL has no construct for exactly specifying the size of data areas or for ascribing structure to these areas. Fields within words must be referenced via shifts and logical operations such as AND and OR.

(ii) The particular programming style of the BCPL SASL interpreter utilises many of BCPL's wide range of control statements. We believe that this detracts from program readability as some BCPL control statements such as UNLESS and TEST we find opaque.

In comparing the inherent suitablity of SUILVEN and BCPL as machine description languages, rather than merely their utility for describing the SASL-machine, two salient factors emerge:-

(i) SUILVEN is a superior language for describing the s-machine data areas because of its declarations which allow the user to exactly specify the size and structure of these areas.

(ii) BCPL is a better language for describing complex s-machine instructions(as is PASCAL), primarily because recursion is a feature of the language. It must be emphasised, however, that BCPL's plethora of control constructs may be abused, producing opaque and unreadable programs.

In summary, therefore, if properly used, BCPL can probably provide a clearer description of abstract machine architecture than SUILVEN, assuming that the BCPL description is supplemented with further information specifying the exact structure of the s-machine data areas. It also assumes that control constructs are used in a disciplined manner - something which is forced on the SUILVEN programmer.

6.7.2 SUILVEN and MIL

In this section, the SUILVEN version of the SASL-machine is compared with the corresponding interpreter encoded in MIL.

Unfortunately, the implementation of this MIL s-machine was delayed by difficulties in implementing the SASL-machine instructions to handle declarations. By the time these difficulties had been resolved, the machine around which our

more general project was based, had changed from the B1700 to the PDP/11, and the MIL implementation of the SASL-machine was never properly tested. However, the program is essentially complete and, by extrapolation, conclusions regarding its performance have been drawn.

As MIL is almost at the level of an assembly language, it is not intended to provide a behavioural description of an s-machine. Our comparison of MIL and SUILVEN is not, therefore, a comparison of the languages as MDL's. Rather, the microcode generated by the SUILVEN compiler is compared with the MIL program. Table 6.3 below summarises some information about the corresponding interpreters.

	SUILVEN	MIL
Number of microinstructions	1890	1320
Number of microinstructions after hand optimisation	1480	1320
Mean number of microinstructions per SASL-machine instruction	46	40
Time taken to execute example program(clock cycles)	6240	3000(estimate)
Time taken to implement interpreter	10 weeks	7 months +

TABLE 6.3

A COMPARISON OF MIL AND SUILVEN INTERPRETERS

Notice that, after hand optimisation, the number of microinstructions in each version of the interpreter is roughly comparable. Notice also that the execution time of the SUILVEN encoded interpreter differs considerably from the estimated time taken by the MIL program. The SASL program used for making this comparison was the following short program:-

```
rec  sumlist (x,y) be
     y = () -> x;
     x + sumlist y
in
     sumlist(1,2,3,4,5,6,7,8,9,10)
```

This program adds lists of numbers - in this case its answer would be 55.

The program was translated to SASL-machine code and executed on our B1700 simulator. The program execution time was estimated for the MIL interpreter by examining which machine instructions were used and then hand simulating the action of the interpreter.

The large discrepancy in execution times may be explained by the fact that the MIL program makes better use of registers and the machine scratchpad. Therefore, far fewer store transfers need take place and the execution time is correspondingly decreased.

For constructing microprograms, SUILVEN's chief advantage is that a working program may be achieved fairly quickly. Although we cannot claim our SUILVEN interpreter to be completely debugged, its implementation time compares very favourably with the time needed to program an interpreter in a low-level language.

6.8 Summary and Conclusions

This chapter has described the programming, in SUILVEN, of two s-machine interpreters. These SUILVEN implementations have been compared with corresponding interpreter implementations in both high and low-level languages. The machines which were implemented are:-

(i) A stack-oriented s-machine for PASCAL

(ii) A list oriented s-machine for SASL

These machines have completely different architectures. Because of this, we reasoned that each machine would test different features of SUILVEN, hence providing a balanced language evaluation.

SUILVEN is an adequate language for describing and evaluating the PASCAL machine. However, because it lacks recursion, its description of some SASL machine instructions is a little unnatural.

It is an excellent language for the description of s-machine data areas - indeed this is probably the best designed feature of SUILVEN. Its facility for exactly specifying the width of dat areas and its flexible structure assignment, provide a clear and readable description of s-machine data organisation. For describing complex recursive machine instructions, the language is inadequate, but it is suitable for describing simpler s-machine operations.

The code generated by the SUILVEN compiler for a typical interpreter seems to contain about 20-25% more microinstructions than the corresponding MIL encoded program. This figure may be significantly reduced by hand optimisation of the generated microcode. The run-time efficiency of a SUILVEN interpreter is much less than that of an interpreter encoded in machine code. This is primarily due to the difficulty of generating microcode which will optimally utilise the fast machine registers.

A full evaluation of SUILVEN and possible improvements in the language, is the subject of the following chapter.

CHAPTER 7

CONCLUSIONS

In this final thesis chapter, we critically examine the work which we have done in the course of the project, and suggest improvements which could be made. We draw a number of conclusions from the research work we have undertaken and put forward some suggestions for further work in this research area.

However, before discussing the above in detail, let us consider our research in the context of the more general research project discussed in chapter 1.

The research which we undertook was part of a larger research project investigating techniques for the construction of abstract machines. As part of this, in parallel with our project, there were a number of other projects investigating machine-dependent microprogramming languages, and language-oriented abstract machines.

When the general project was initially conceived, it was decided to base the system implementation on the B1700 range of computers, for the reasons discussed in chapter 1. As we were involved in the early stages of the project, it was natural that out own research should be biased towards this machine.

As economic circumstances changed, the possibility of acquiring a B1700 computer receded. Dedicated hardware was considered essential to the success of the main research project, and a

decision was made to acquire an alternative,cheaper machine - a PDP/11.

By this time, we were committed to our B1700 based project and we decided to continue our research towards some sort of conclusion. Unfortunately, the change in machine from the B1700 meant that some of the support we had hoped for was redirected towards the alternative machine. In particular, the MIL version of the SASL machine was never completely operational and a MIL version of the PASCAL machine was never even started. Had these programs been available, we believe that we could have better evaluated the work we have done.

In an attempt to evaluate the success of the project, we shall examine how well we achieved the aims of the research, as set out in chapter 1. To recap, the aims of our project were:-

(i) To construct a simulator for the B1726 system at the microprogramming level. The simulator was to be implemented on an IBM 360/44 computer.

(ii) To design and implement a high-level microprogramming language for the B1700 range of machines. Not only should this language be compilable into B1700 microcode, but also,a program listing should constitute a clear description of the implemented abstract machine architecture.

(iii) To evaluate the utility of this microprogramming language by comparing s-machine implementations in this language with the same machine implementations in other programming languages.

After considering our success in achieving the above aims, we draw a number of more general conclusions concerning the way in which we approached the problem. We then suggest an alternative approach, which, in the light of experience, we now believe would have been better and speculate upon future research work in the field of language-oriented machine implementation.

In the sections below, we consider aims (i) - (iii) in turn and critically evaluate the work done towards these aims. Where appropriate, we indicate shortcomings in our design and methods, and suggest modifications to alleviate these shortcomings.

7.1 The B1726 Simulation

This section of the project involved constructing a simulator for the B1726 computer and a compiler for the standard machine microprogramming language, MIL. These programs were written in ALGOLW and SNOBOL4 respectively, according to the specification laid out in the appropriate Burrough's reference manuals.

The MIL compiler which we developed, performs satisfactorily and the processor simulator is an accurate simulation of the micro-architecture af the B1726 computer.

Unfortunately, the simulator which we developed is not an exact replica of the B1726 system as seen by a microprogram executing on that machine. On the real machine, executing microprograms are given considerable support by the machine operating system and it did not prove possible to simulate this support. There were two reasons for this:-

(i) We anticipated that simulating the operating system actions would impose unacceptable overheads in the time taken to simulate a microprogram execution. We also believed that the size of the simulator would be significantly increased.

(ii) The documentation which was available to us concerning the role of the operating system was incomplete and imprecise. To fully understand that role would have involved a good deal of work, studying the source code listing of the operating system. We did not consider the results important enough to justify this work.

As discussed in chapter 3, which describes the simulation programs, the simulation of a machine such as the B1700 under a batch environment poses problems. Features such as external interrupts, console communication, and user interaction cannot be properly simulated.

In spite of these drawbacks, we believe that there were a number of benefits gained from undertaking this part of the project.

These benefits were:-

(i) The construction of the simulator in the initial stages of the project ensured that we acquired an intimate knowledge of the micro-architecture of the B1700. This knowledge proved invaluable when implementing the SUILVEN compiler.

(ii) The simulator can act as a test bed for locally written microprograms. Using the simulator, it is possible to test much of the logic of these programs, and their subsequent implementation on a real B1700 merely involves modification of input/output and interrupt handling routines.

In general, we consider that the exercise of constructing the simulation system was worthwhile, but that such a system is, in no way, an adequate substitute for a real machine.

7.2 SUILVEN as a Machine Description and Implementation Language

The high-level language SUILVEN was designed to serve a dual purpose:-

(i) To describe the architecture of abstract machines

(ii) To implement abstract machine interpreters as microprograms on the B1700 computer.

We shall evaluate how well SUILVEN meets these design aims and shall examine shortcomings in the design of the language. These became apparent when SUILVEN was used to implement s-machines for SASL and PASCAL.

The description of an abstract machine must contain a complete specification of the data areas of that machine, both in terms of their width(in bits) and their structure. The data description facilities offered by SUILVEN, that is, the BITS declaration, the TEMPLATE and DEFINE declarations, and the REDIMENSION declaration, ensure that an accurate description of data areas is possible. Not only may the width of a data area be precisely specified, but also the structure of that area may be varied depending on the instruction operating on it.

However, as well as defining the size and structure of machine data areas, we have now reached the conclusion that an abstract machine description language ought to give some indication how these are to be mapped onto the data areas of the underlying machine. For example, when implementing a stack machine, the stack pointer will be frequently accessed, and should be retained in a fast access register. When the variable STACK_POINTER is declared in the program, it should be possible to specify this. For example:-

```
BITS(24) REGISTER  STACK_POINTER;
```

This would declare that the variable STACK_POINTER should be stored in a machine register.

Similiarly, in order that conditions such as interrupts, overflow, etc may be specified, a description language should have a declaration setting up single bit boolean flags. Naturally, there must also be operators to manipulate these objects.

Whilst SUILVEN has a flag declaration, the register declaration is implemented by means of a compiler directive. We now believe that this is an essential part of machine specification, rather than a compiler feature, and ought to be included in a machine description language.

Apart from this single exception, we are satisfied with SUILVEN's data description features, and have found them to be both convenient and adequate for describing abstract machine data areas.

As well as providing a description of machine data areas, a description language must also specify the operation of the machine instructions. It is this description which is mapped onto the microcode of the underlying machine.

The operators presently provided in SUILVEN were found to be adequate for describing and implementing the PASCAL s-machine. However, the more complex SASL machine, some of whose instructions are at a higher level than PASCAL machine instructions, highlighted significant gaps in the operation repertoire provided by SUILVEN.

The features lacking in the language were concerned with the manipulation of structured data areas and with the implementation of highly recursive, high-level machine instructions.

7.2.1 Structured Data Operations

The lack of provision of operations which deal with structured data areas as a whole, rather than field by field, caused some inconvenience in describing and implementing the SASL s-machine. The following example, taken from that machine description, illustrates how three separate statements must be made to assign values to the fields of a structured data area:-

```
LIST(.NEWCELL.).TAIL := TOPS_DATA;
LIST(.NEWCELL.).TAG := SECS_TAG;
LIST(.NEWCELL.).HEAD := SECS_DATA;
```

The code generated for these rather repetetive assignment statements is inefficient, as the address of each field is completely recomputed for each assignment. The B1700 has an automatic address-increment feature which updates the memory address register in parallel with a memory fetch. Therefore, as well as simplifying machine description, operations on structured data areas would provide enough information for the compiler to use this facility.

There are two possible ways of including this feature in a machine description language:-

Simultaneous Assignment

An assignment statement which may be used to assign to all fields of a structured data area should be a feature of a machine description language. For example, the assignments above could be encoded:-

```
LIST(.NEWCELL.) := 0, SECS_TAG, SECS_DATA, TOPS_DATA;
```

A list element is structured into four fields and the expressions on the right side of the assignment are assigned to each field in turn.

A WITH Statement

This statement would be used for operations on more than one, but not all fields of a structured variable. The suggested construct is based on the PASCAL WITH statement, designed for working with records. This allows the user to specify the name of a structure and, in the succeeding block, to refer directly to the fields of that structure rather than prefix these names with the structure name. For example:-

```
WITH  LIST(.NC2.) DO
    £(
        TAG := LIST(.TOPS_DATA.).TAG;
        HEAD := SECS_DATA;
    £)
```

This would specify that assignments be made to the TAG and HEAD fields of LIST(.NC2.). We anticipate that efficient microcode, using auto-address updating, could be generated for this construct.

7.2.2 Recursive Machine Instructions

Some of the machine instructions of the SASL machine are best described and implemented recursively. In particular, the instructions for altering the environment list fall into this category.

The present SUILVEN implementation of procedures was not designed to allow recursion. Although a procedure may call itself, by virtue of the fact that an entry is made in the appropriate compiler table when a procedure heading is processed, the saving and restoring of local variables must be handled by the SUILVEN programmer. Mutual recursion is not possible, as names must be declared before use. These restrictions added to the complexity of the SASL machine description and circumventing them introduced redundancies in the generated microcode.

Unfortunately, it is not possible to make a relatively simple modification to SUILVEN which would permit recursion and mutual recursion. For efficiency reasons, we are convinced that s-machine storage allocation should be static, whereas the implementation of recursion requires a dynamic storage allocation scheme.

We see no way to resolve this dilemma, apart from separating the machine description from the machine implementation. A machine description language should allow recursive descriptions where this is natural, yet an implementation language should not. This is discussed in more detail in section 7.4, which deals with our recommendations regarding the description and implementation of abstract machines.

To sum up, therefore, SUILVEN has shown itself to be a suitable vehicle for describing the data areas of abstract machines. We believe that it is superior to other machine description languages and general purpose programming languages for this function.

The operations provided by SUILVEN are adequate for describing machine instructions with low semantic content, such as are used in the PASCAL machine. However, because of the lack of recursion and the lack of structured data operations, SUILVEN is not completely satisfactory for describing higher level machines such as the SASL machine. Whilst operations could be added to the language which would operate on structured data areas, it is not possible to add recursion without introducing a dynamic storage allocation strategy. We believe this would compromise efficiency to such an extent as to be unjustifiable.

7.3 Comparison of Abstract Machines

The third aim of our project, was to construct abstract machines using SUILVEN and compare them with the same s-machines implemented using other programming languages. As explained in the introduction to this chapter, some problems were encountered in achieving this aim, due to the change in machine of the more general research project.

As discussed in the previous chapter, s-machines were programmed in SUILVEN for implementations of PASCAL and of SASL. These s-machines were compared with similiar machines programmed in both low and high-level languages. Our conclusions on these comparisons are discussed fully in chapter 6, with the main points summarised below:-

(i) SUILVEN is a better language than both PASCAL and BCPL for describing the data organisation of abstract machines. BCPL's lack of structured data types and, conversely, PASCAL's very strict type discipline meant that structured data areas and areas of variable type(such as a stack) were described in an unnatural fashion in both PASCAL and BCPL.

(ii) For the fairly low-level PASCAL machine, the SUILVEN and PASCAL descriptions of the machine instructions were almost equivalent both in size and in clarity. PASCAL's operations such as set operators made for conciseness in describing their own implementation but gave no indication

how they were to be implemented.

BCPL was better than SUILVEN for describing the operation of the higher-level SASL machine. This was a result of SUILVEN's lack of structured data operations and recursion.

(iii) Microprogrammed s-machines generated by the SUILVEN compiler appear to contain about 25% more microinstructions than the equivalent hand-coded programs. This figure may be reduced by hand-optimising the generated microcode. Because of the difficulty of optimising the usage of the B1700 scratchpad, SUILVEN programs execute at about half the speed of hand-coded programs.

This execution time may be significantly reduced if the microcode generated by the compiler is hand-optimised. We estimate that execution speeds may be doubled by dint of relatively simple hand-optimisation.

We believe that constructing abstract machines in SUILVEN and hand-optimising these machines can be achieved in a significantly shorter time than constructing the same machines in a low-level language.

The above conclusions show that SUILVEN s-machines are less efficient than machines programmed in a low-level language. They also show that SUILVEN is not completely satisfactory as a vehicle for describing machine architecture. As a result, we no longer believe that our approach of developing a compromise machine description/implementation language is viable.

7.4 General Conclusions

In this section of the thesis, we attempt to draw a number of general conclusions derived from our work on tools for the implementation of abstract machines. These conclusions reflect our belief that the project was worthwhile, and that the problem of efficiently implementing and documenting abstract machines is by no means solved. As discussed below, we believe that an alternative approach involving separate machine description and implementation languages may be more fruitful.

We have derived four general conclusions from our work:-

(i) The B1726 simulator which we produced is, to a limited extent, successful but,because of three main factors, is not an adequate substitute for a real machine. These factors are:-

(a) The system is slow.

(b) The simulator lacks an operating system.

(c) The user cannot interact with the simulation.

We believe that a research project such as ours cannot be satisfactorily undertaken without a real, dedicated machine. This machine must be user microprogrammable. In the forseeable future, microprogramming will be the most cost-effective technique of implementing abstract machines and, as micro-architecture is significantly different from standard Von Neumann or high-level machine architectures, research into abstract machine design and construction should be carried out on a microprogrammable machine.

(ii) SUILVEN is a reasonable tool for implementing abstract machines in a research environment. In this type of situation, although the generated microcode is less efficient than that encoded by hand, SUILVEN's advantages, viz programs may be produced quickly and easily modified, justify its use. Using a high-level microprogramming language means that different s-machine designs may be constructed and evaluated without undue programming cost.

Because of its inefficiencies, SUILVEN is not suitable for use in a production environment.

(iii) As a machine description language, SUILVEN is adequate for describing the architecture of low-level abstract machines. In particular, its features for describing machine data areas are superior to those in other machine description or general purpose languages which we have examined.

The language is not completely adequate for describing machines with high-level operations, particularly if these are recursive operations or operations on structured data areas. SUILVEN lacks constructs to describe such operations.

(iv) From the above, we conclude that, if microprogrammed s-machines are to be widely implemented and used, there must be a change of strategy. It appears that the requirements of a machine description and a machine implementation language are different. A machine description language

must express, with the utmost clarity, the structure of the machine data areas and operations whereas, machine implementation languages must permit the user to have control over the microcode generated by the compiler.

We believe, therefore, that developments in machine description languages and machine implementation languages must proceed in parallel. Future MDL's will be high-level languages with rigidly enforced conventions which will force the user to produce clear and readable documentation. Possibly they will be translatable into other high-level languages so that a machine simulator may be produced from the machine description.

Machine implementation languages, on the other hand, will be higher-level machine-oriented languages in the manner of Wirth's PL360(W7). These will retain much of the convenience of high-level microprogramming yet give the programmer full control over the microinstructions generated by the compiler.

In the light of these conclusions, we recommend that the continuing development of SUILVEN be abandoned and that future developments proceed using separate machine description and implementation languages. Should this approach be adopted, we believe that flexibility can be retained yet microcode, efficient enough to meet engineering and economic requirements, might be produced.

7.5 Future Research

In the near future(5-10 years) we envisage that developments in this field will proceed using high-level description languages and machine oriented microprogramming languages. With our present expertise, such languages may be fairly easily implemented, although problems of generating absolutely optimal code must be overcome before such an approach will completely satisfy assembly language adherents.

We also believe that there is potential for considerable automation in the construction of s-machine interpreters. It appears that all interpreters have the same fundamental structure and, for many applications, this may be programmed automatically. We envisage a system being constructed which will enable the user to tailor this general purpose interpreter structure to his own needs. Indeed, the author of this dissertation is presently designing and constructing an interactive system for this purpose(S3).

Developments in machine oriented languages and design automation appear to be the most fruitful avenues of research in the near future. On a longer term basis we forsee other developments occurring. These are detailed overleaf.

(i) Machine independent microprogramming languages which are optimally efficient will become possible. Such languages require micro-architectures designed for their support but we believe that, as microprogramming becomes more widespread, these architectures will evolve. There is a need for research into micro-architectures needed to support high-level microprogramming languages.

(ii) In conjunction with the development of microprogramming techniques, we forsee the s-machine level becoming obsolete. High-level languages will be directly executed. Some research towards this end has been deseribed by Laliotis(L6) who discusses the architecture of the SYMBOL computer. There is considerable potential for research in this field which will involve a high level of cooperation between digital engineers and software engineers.

We do not envisage microprogramming being eclipsed as an implementation tool in the near future. It will not be replaced until an automatic machine construction technique has evolved, which has greater convenience, economy and flexibility.

REFERENCES

B1. C.G. Bell and A. Newell
Computer Structures : Readings and Examples
pub. Mcgraw-Hill, New York, 1971

B2. Burroughs Corporation
B1700 Systems Reference Manual
Detroit, 1972

B3. Burroughs Corporation
Micro-Implementation Language(MIL) Reference Manual
Detroit, 1972

B4. C. Bohm and G. Jacopini
Flow Diagrams, Turing Machines and Languages with only Two Formation Rules
Comm. ACM 16, 7(July,1973), 443-454

D1. J. Darringer
A Language for the Description of Digital Computers
Proc. Design Automation Workshop, 1968, 15-1 - 15-8

D2. D.J. Dewitt, M.S. Schansher, and D.E. Atkins
A Microprogramming Language for the B1726
Sixth Annual Workshop on Microprogramming, ACM, Sept.1973
21-29

D3. E. Dijkstra
GO TO Statement Considered Harmful
Comm. ACM 11, 3(March,1968), 147-148

D4. E. Dijkstra
Guarded Commands, Non-determinacy and Formal Derivation of Programs
Comm. ACM 18, 8(August,1975), 453-457

D5. O.S. Dahl, E. Dijkstra, and C.A.R. Hoare
Structured Programming
pub. Academic Press, London, 1970

D6. D.J. Dewitt
Extensibility - A New Approach for Designing Machine Independent Microprogramming Languages
Ninth Annual Workshop on Microprogramming, ACM, Sept.1976 33-42

E1. R.H. Eckhouse
A High-Level Microprogramming Language(MPL)
Ph.D. Thesis, Dept. of Comp. Sci., State Univ. of New York at Buffalo, June 1971

F1. R.N. Fisher, G. Mcquarrie, and R. Morrison
A Microprogramming Language for the B1700 Computer
TR/75/9, Dept. of Comp. Sci., Univ. of St Andrews, February 1975

F2. R.W. Floyd
Syntactic Analysis and Operator Precedence
J. ACM 10, 7(July,1963), 316-328

G1. R. Griswold
The Macro Implementation of SNOBOL4
pub. W.H. Freeman, San Francisco, 1972

G2. D. Gries
Compiler Construction for Digital Computers
pub. Wiley, New York, 1971

G3. R. Griswold, J.F. Poage, and I.P. Polonsky
The SNOBOL4 Programming Language
pub. Prentice-Hall, New Jersey, 1971

H1. F.J. Hill and G.R. Peterson
Digital Systems : Hardware Organisation and Design
pub. Wiley, New York, 1973

H2. D.A. Huffman
A Method for the Construction of Minimum Redundancy Codes
Proc. IRE 40(Sept,1952), 1098-1101

I1. K.E. Iverson
A Programming Language
pub. Wiley, New York, 1962

I2. K.E. Iverson
A Common Language for Hardware, Software, and Applications
Proc. FJCC, Philadelphia, Dec. 1962, 121-129

I3. K.E. Iverson
Programming Notation in Systems Design
IBM Systems Journal, June 1963, 117-127

I4. IBM Corporation
System 360 : Principles of Operation
Poughkeepsie, N.Y., 1964

J1. K. Jensen, K.V. Nori, U. Amman, and H.H. Nageli
Implementation Notes for PASCAL
Technical Report No. 10, Eidgenossische Technische Hochschule, Zurich

K1. D. Knuth
An Empirical Study of FORTRAN Programs
Software - Practice and Experience 1, 2(March,1971), 105-133

L1. W. Lonergan and P.King
The Design of the B5000 System
Datamation 7, 5(May,1961)

L2. P.J. Landin
The Mechanical Evaluation of Expressions
Comp. J. 6, 2(April,1964), 308-320

L3. H.W. Lawson Jnr. and L.Blomberg
The Datasaab FCPU Microprogramming Language
Proc. Sigplan/Sigmicro Interface Meeting, May 1973, 86-96

L4. H.W. Lawson Jnr. and B.K. Smith
Functional Characteristics of a Multi-Lingual Processor
IEEE Trans. Comput. C-20(July,1971), 732-742

L5. H.W. Lawson Jnr.(Chairman)
Discussion on Microprogramming Languages
Proc. Sigplan/Sigmicro Interface Meeting, May 1973, 175-181

L6. T.A. Laliotis
The Architecture of the SYMBOL Computer System
High-Level Machine Architecture, ed. Y. Chu,
pub. Academic Press, London, 1975

M1. W.M. McKeeman, J.J. Horning, and D.Wortman
A Compiler Generator
pub. Prentice-Hall, New Jersey, 1970

N1. G. Nelson and D.Turner
A Microprogrammed SASL Interpreter
TR/75/5, Dept. of Comp. Sci., Univ. of St Andrews, March 1975

O1. E.I. Organick
Computer Systems Organisation
pub. Academic Press, London, 1973

O2. D.R. Oestricher
A Microprogramming Language for the MLP-900
Proc. Sigplan/Sigmicro Interface Meeting, May 1973, 113-119

P1. P.C. Poole,W.M. Waite, and M.C. Newey
Abstract Machine Modelling to Produce Portable Software -
A Review and Evaluation
Software - Practice and Experience 2, 2(March, 1972), 107-136

P2. D.L. Parnas
A Language for Describing the Functions of Synchronous Systems
Comm. ACM 9, 2(February,1966), 72-77

R1. B. Randell and L.J. Russell
ALGOL60 Implementation
pub. Academic Press, London, 1964

T1. D. Turner
The SASL Language Manual
CS/75/1, Dept. of Comp. Sci., Univ. of St Andrews, Jan. 1975

T2. D. Turner
An Implementation of SASL
TR/75/4, Dept. of Comp. Sci., Univ. of St Andrews, March 1975

S1. I. Sommerville
A Simulator for the B1700
TR/75/6, Dept. of Comp. Sci., Univ. of St Andrews, April 1975

S2. I. Sommerville
An MIL Translation System
TR/75/7, Dept. of Comp. Sci., Univ. of St Andrews, April 1975

S3. I. Sommerville
The Automatic Implementation of Interpreters
T.R. 4/77/11, Dept. of Comp. Sci., Heriot-Watt University, Edinburgh, February, 1977

W1. N. Wirth
The Programming Language PASCAL
Acta Informatica 1, 1(1971), 35-63

W2. D. Wortman
Language-Oriented Machines
CSRG-20, Comp. Sys. Res. Group, Univ. of Toronto, Dec. 1972

W3. W.T. Wilner
The Design of the B1700
Proc. FJCC, Anaheim California, 1972, 489-497

W4. W.T. Wilner
Burroughs B1700 Memory Utilisation
Proc. FJCC, Anaheim California, 1972, 579-586

W5. W.T. Wilner
Microprogramming Environment on the Burroughs B1700
COMPCON 79, Digest of Papers, IEEE(September, 1972), 103-106

W6. N. Wirth and H. Weber
Euler : A Generalisation of ALGOL, and its Formal Definition
Comm. ACM 9, 1(Jan.,1966), 13-23
Comm. ACM 9, 2(Feb.,1966), 89-99

W7. N. Wirth
PL/360 : A Programming Language for the 360 Computers
J. ACM 15, 1(Jan.,1968), 37-74

W8. B.A. Wichmann
ALGOL 60 : Compilation and Assessment
pub. Academic Press, London, 1973

W9. D. Wortman
Six PL/1 Compilers
Software - Practice and Experience, 6, 2(March,1976), 38-45

APPENDIX 1

A DESCRIPTION OF THE MICROPROGRAMMING LANGUAGE SUILVEN

HERIOT-WATT UNIVERSITY, EDINBURGH

DEPARTMENT OF COMPUTER SCIENCE

TR1/77/8

SUILVEN

LANGUAGE REFERENCE MANUAL

IAN SOMMERVILLE

FEBRUARY, 1977.

1.0 INTRODUCTION

This manual is a definitive description of the programming language SUILVEN, a high-level microprogramming language for the B1700 series of computers. The language was based on a soft machine description language, described by Sommerville (1) and its design pholosophy and implementation are also described by that author (2). We assume the reader has some knowledge of programming language terminology.

2.0 NOTATION

The notation used to describe the syntax of SUILVEN is an extended form of BNF. The symbols <,>, ::=, and | have their usual meanings but we have extended BNF by introducing a starred square bracket construct, [ ]*. The enclosure of an element in starred square brackets means that an occurrence of that element may be repeated zero or more times.

3.0 BASIC SYMBOLS

A SUILVEN program consists of a sequence of identifiers, constants and special symbols. These are syntactically defined below:

<identifier> ::= <letter> [<idsymb>]*

<idsymb> ::= <letter> | <digit> | _

<letter> ::= A -- Z

<digit> ::= 0 -- 9

<identifier list> ::= <identifier> [, <identifier>]*

<constant> ::= <number>|<binconst>|<hexconst>

<binconst> ::= % <bindigit>

<hexconst> ::= # <hexdigit>

<bindigit> ::= 0 | 1

<hexdigit> ::= 0 -- 9 | A -- F

<number> ::= <digit> [<digit>]*

<special symbol> ::= <reserved word> | <char symbol>

<char symbol> ::= , | . | (|) | (. | .) | := |
> | < | = | >= | <= | ¬= | ¬ | & |
<or> | : | ; | ' | + | -| * | / | £(| £)

<or> ::= | where | is not a metasymbol

<reserved word> ::= REM | SHL | SHR | XOR | MACRO | BITS|
ARRAY | TEMPLATE | DEFINE | FLAG|
PROCEDURE | FUNCTION | IF | THEN | ELSE|
WHILE | DO | REPEAT | UNTIL | CASE|
OF | ENDCASE | END | TRUE | FALSE|
SET | UNSET | READ | WRITE | NULL|
EXIT | STOP | REDIMENSION|
ENDPROGRAM

<string> ::= '<chars>'

<chars> ::= <chars> [<char>]*

<char> ::= <letter> | <digit> | <blank> | , | . | (|) |
+ | ¬ | * | / | > | < | = | - | & | <or> | : |
; | % | # | @ | $ | £

<blank> ::= *A space*

4.0 PROGRAM STRUCTURE

A SUILVEN program can be considered as consisting of three logical sections. Firstly, a set of declarations establishing the names of variables, structures, macros etc, secondly a set of procedure declarations and finally a main program consisting of one or more SUILVEN statements. A program is terminated by the reserved word ENDPROGRAM.

Syntax

```
<SUILVEN program> ::= <declaration part>
                      <procedure declaration part>
                      <statement part> ENDPROGRAM
```

5.0 DECLARATIONS

The declaration part of a SUILVEN program consists of a series of declarations of macros, bits variables, structures, and flags.

Syntax

```
<declaration part> ::= <declaration>[;<declaration>]*

<declaration>      ::= <macro declaration>|
                       <bits declaration>|
                       <template declaration>|
                       <define declaration>|
                       <flag declaration>
```

5.1 MACROS

Macro declarations define equivalent strings of characters.

Syntax

<macro declaration> ::= MACRO <string> = <string>

Semantics

Each occurrence of the string defined on the left side of the equals sign above is replaced by its corresponding right side throughout the SUILVEN program.

5.2 BITS DECLARATIONS

Bits declarations in a SUILVEN program name storage areas.

Syntax

<bits declaration> ::= <simple bits declaration>|<bits array declaration>

<simple bits declaration>::= BITS (<number>) <identifier list>

<bits array declaration> ::= BITS (<number>) ARRAY (<number>)
<identifier list>

Semantics

Bits declarations associate a name with a machine storage area of a specified width. The number specified after the reserved word BITS specifies the width of that area. In an array declaration, the number following the reserved word ARRAY specifies the number of elements in the array. The first element of the array is considered to be element 1.

5.3 TEMPLATE DECLARATIONS

Template declarations serve to define a structure, and associate a name with that structure.

Syntax

<template declaration> ::= TEMPLATE <identifier> =
<field> [,<field>]*

<field> ::= <identifier> (<number>)

Semantics

A TEMPLATE declaration associates the name on the left side of the equals sign above with a sequence of one or more fields. A field consists of a name followed by a bracketed number, which specifies the width of that field in bits.

5.4 DEFINE DECLARATIONS

Define declarations serve to associate a data area with a structure.

Syntax

<define declaration> ::= DEFINE <identifier>:<identifier list>

Semantics

The identifier on the left side of the colon should be a previously declared template name and the identifier list should consist of names of previously declared data areas. The define declaration specifies that each of these data areas should be considered to have the structure defined by the template name. Notice that the width in bits of the data areas should be equal to the cumulative widths of the template fields.

5.5 FLAG DECLARATIONS

Flag declarations establish names to be associated with logical variables.

Syntax

```
<flag declaration>      ::=  FLAG <identifier list>
```

Semantics

Each identifier in the identifier list is considered to be a logical, 1-bit variables which may take the truthvalues TRUE or FALSE.

6.0 PROCEDURE DECLARATIONS

Procedure and function declarations establish names which are associated with SUILVEN declarations and statements.

Syntax

```
<procedure declaration>  ::=  <function declaration> | <proper procedure
                                                        declaration>

<function declaration>   ::=  FUNCTION <procedure header><procedure body>

<proper procedure declaration>::= PROCEDURE <procedure header><procedure body>

<procedure header>       ::=  <identifier> (<formal parameter list>)

<formal parameter list>  ::=  <bits declaration> [;<bits declaration>]*
                              |  <empty>

<procedure body>         ::=  <local declaration part>

                              <statement part>

                               END <result part>

<local declaration part> ::=  <empty> | <local declaration> [;<local
                                                        declaration>]*

<local declaration>      ::=  <declaration> | <redimension declaration>
```

<redimension declaration> ::= REDIMENSION <redimension list>

<redimension list> ::= <redimension> [,<redimension>]*

<redimension> ::= <identifier> (<number>, <number>)

<result part> ::= <empty> |=<expression>

Semantics

A procedure declaration associates a name with a sequence of SUILVEN declarations and statements. Procedures may be either function procedures or proper procedures. Function procedures always return a bitstring as a result, this bitstring being specified as the expression in the result part above.

Both function procedures and proper procedures may have zero or more formal parameters. A formal parameter is a bits variable which may be referred to within the procedure. Its scope is local to that procedure i.e. it may only be referred to within the procedure body. An initialisation is associated with each formal parameter each time a procedure is activated.

The local variables declared within a procedure establish names which are only in scope within the procedure. These names may be used in an identical fashion to global names, in SUILVEN statements.

The REDIMENSION declaration is only meaningful within procedures and it serves to restructure a global array. The array name is specified followed by a bracketed pair of integer constants. The first of these specifies a new length for the array and the second the width of each element in the new array. Notice that the product length and width should be less than or equal to the product length and width in the global array declaration. The array specifications revert to their global specifications on exit from a procedure where the array is redimensioned.

7.0 SUILVEN STATEMENTS

This section describes executable SUILVEN statements of which there are three basic types - assignments, procedure calls and control statements.

Syntax

<statement part> ::= <statement> [;<statement>]

<statement> ::= <assignment> | <procedure call> | <control statement> | <compound statement>

7.1 EXPRESSIONS

The expression is a basic part of most other SUILVEN statements. It may be evaluated to return a bitstring.

<expression> ::= <signed var> [<operator> <signed var>]*

<signed var> ::= <unary operator> <var>

<var> ::= <identifier> | <constant> | <function designator> | <indexed var> | <array index>

<function designator> ::= <name>(<actual parameter list>)

<actual parameter list> ::= <empty> | <expression list>

<expression list> ::= <expression> [,<expression>]*

<unary operator> ::= <empty> | ¬

<indexed var> ::= <structured element> . <identifier>

<structured element> ::= <identifier> | <array index>

<array index> ::= <identifier> (.<expression>.)

operator ::= +|-|+|/|&|<or>|SHL|SHR|XOR|REM|

Semantics

An expression consists of a series of one or more elements which may be evaluated to produce a result which is a bitstring.

Each element in the expression is evaluated on a left to right basis and operators, where they are included, are applied on the same basis. No operator takes precedence over another.

The basic components of an expression may be the name of a bits variable, an element of an array, a field of a structured bits variable or array element or a function designator which specifies that the named function be evaluated.

The operators +, -, and *, have their usual meanings, / is an integer division operator and REM is the remainder operator. The shift operators SHL and SHR shift the bits of their left hand operand either to the left or the the right by the number of bits specified in their right hand operand. The operator & is a logical AND, | is a logical inclusive OR and XOR is a logical exclusive OR operator.

A expression may be negated by preceding it by the unary operator ¬ .

7.2 LOGICAL EXPRESSIONS

Logical expressions are a basic constituent of SUILVEN control statements.

<logical expression> ::= <relation> [<logical connective> <relation>]*

<relation> ::= <expression> <relation operator> <expression>

<logical connective> ::= AND | OR

<relation operator> ::= =| ¬ =| > | >=| < | <=

Semantics

A logical expression is evaluated to produce a truthvalue, TRUE or FALSE. It consists of one or more relations, which themselves are evaluated to truthvalues, connected by the logical connectives AND or OR. AND has its obvious meaning, OR is an inclusive OR.

A relation is a comparison of two expressions. This comparison returns a truthvalue and may be made on the basis of equality (=), inequality (¬=), greater than (>), greater than of equals (>=), less than (<), or less than/equals (<=).

7.3 THE ASSIGNMENT STATEMENT

The SUILVEN assignment statement has the same form as pertains in most other high-level languages.

Syntax

<assignment> ::= <lhvar>:= <expression>

<lhvar> ::= <identifier> | <array index> | <structured element>

Semantics

The semantics of the assignment statement are well known. The expression to the right of the composite symbol := is evaluated and that value is assigned to the element on the left of :=.

7.4 PROCEDURE CALLS

Again SUILVEN procedure calls are similar to procedure calls in other high-level languages.

Syntax

<procedure call> ::= <identifier>(<actual parameter list>)

Semantics

When a procedure call is encountered, the parameters if any, are evaluated. That value is then used to initialise the procedure formal parameters. Call by value is the only parameter passing mode available in SUILVEN.

After initialising the formal parameters, control is transferred to the code in the named procedure which is then executed. After execution, control is returned to the SUILVEN statement following the procedure call.

7.5 CONTROL STATEMENTS

SUILVEN has a simple, sparse but adequate set of control statements.

Syntax

```
control statement ::= <if statement>|
                      <while statement>|
                      <repeat statement>|
                      <exit statement>|
                      <stop statement>|
                      <case statement>
```

7.5.1 THE IF STATEMENT

This is the familiar two armed conditional which is available in most high-level programming languages.

Syntax

<if statement> ::= <if clause> <simple statement>

<if clause> <simple statement> ELSE <statement>

<if clause> ::= IF <logical expression> THEN

<simple statement> ::= *Any SUILVEN statement apart from an IF statement.*

Semantics

The logical expression following IF is evaluated. If it is true, the simple statement following THEN is executed and after execution control is transferred to the next SUILVEN statement in the program.

If the result of the logical expression is false, and there is no ELSE part, control is transferred directly to the next SUILVEN statement in the program. If there is an ELSE part, the code following ELSE is executed and control then is transferred to the next program statement.

7.5.2 THE WHILE STATEMENT

This is the familiar looping construct.

Syntax

<while statement> ::= WHILE <logical expression> DO <statement>

Semantics

The logical expression following WHILE is evaluated. It it is true, the statement following DO is executed. The execution sequence then loops so that the logical expression is again evaluated. Again, if true the statement following DO is executed. This sequence continues until the logical expression becomes false, whence control is transferred to the next SUILVEN statement in the program.

7.5.3 THE REPEAT STATEMENT

The repeat statement is designed so that the test for loop exit may be placed anywhere within the loop.

Syntax

<repeat statement> ::= REPEAT <statement> UNTIL <logical expression>
DO <statement>

Semantics

The statement following REPEAT is executed. The logical expression following UNTIL is then evaluated and, if false, the statement following DO is executed. Control then returns to the statement following REPEAT and the process continues until the logical expression is true. Control is then transferred to the next SUILVEN statement.

Either statement in the repeat statement may be null (represented by the reserved work NULL), hence allowing the test for loop exit to be placed at the beginning, in the middle or at the end of a loop.

7.5.4 THE EXIT STATEMENT

This statement permits the programmer to specify immediate return from a procedure.

Syntax

<exit statement> ::= EXIT <result>

Semantics

If a result is specified it is evaluated. Control is then transferred to the statement immediately following the call of the procedure containing the exit statement.

7.5.5 THE STOP STATEMENT

This statement permits the programmer to abort his program.

Syntax

<stop statement> ::= STOP

Semantics

Causes immediate program termination.

7.5.6 THE CASE STATEMENT

This statement allows the programmer to select one of a number of alternatives for execution.

Syntax

<case statement> ::= CASE <expression> OF <statement list>
ENDCASE

<statement list> ::= <statement> [;<statement>]*

Semantics

The expression following the word CASE is evaluated. The statements in the statement list can be considered as being implicitely numbered from 1 to n, where there are n statements in the list. The statement whose implicit number corresponds to the expression value is executed. Control is then transferred to the next SUILVEN statement in the program.

Should the expression value be <1 or >n, where there are n statements in the list, the action of the case statement is undefined.

8.0 STANDARD PROCEDURES

All input/output and flag operations in SUILVEN is carried out using standard procedures.

8.1 INPUT/OUTPUT

SUILVEN has a primitive set if I/O functions which allow the system to accept card input and produce line printer output. These functions are:-

(i) READ ( )

Reads a card image from the input stream into a predeclared buffer called INPUT_BUFFER. Associated with this buffer is a pointer called INPUT_POINTER.

(ii) WRITE ( )

Writes a line image to the printer of a predeclared buffer called OUTPUT_BUFFER. Associated with this buffer is a pointer called OUTPUT_POINTER. After output, OUTPUT_BUFFER is cleared to blanks.

(iii) PUT (<expression>)

Evaluates the expression, converts the result to EBCDIC and moves this result to INPUT_BUFFER (.INPUT_POINTER.)

(iv) PUTSTRING (<string>)

Moves the specified string to INPUT_BUFFER (.INPUT_POINTER.)

8.2 FLAG OPERATIONS

There are four primitive procedures for operating on flag type variables:-

(i) SET (<flag name>)

Sets specified flag to true

(ii) UNSET (<flag name>)

Sets specified flag to false

(iii) TRUE (<flag name>)

Returns value true if flag is set, else false

(iv) FALSE (<flag name>)

Returns value true if flag is unset, else false.

9.0 COMPOUND STATEMENTS

Any group of SUILVEN statements may be converted to a compound statement by enclosing the statements in compound brackets £(and £).

10.0 USING SUILVEN

SUILVEN is available on the IBM 370/168 at the University of Cambridge. It may be accessed via a dial-up terminal using the following Phoenix commands:-

```
SET   your user parameters

C IS10.P: SUILVEN      IF = <input source file name>,
                       OF = <output spool file>,
                       OC = <output code file>
```

The microcode generated by the SUILVEN compiler may be executed on a B1700 simulator using the following Phoenix command:-

```
C IS10.P: B1700SIM      IC  =  <input code file>,
                        OF  =  <output spool file>
```

REFERENCES

(1) A Soft Machine Description Language
I. Sommerville
Dept. of Computer Science, University of St. Andrews
TR/75/6, March 1975.

(2) An Experiment in High-level Microprogramming
I. Sommerville
Ph.D. Thesis, University of St. Andrews, May 1977.

APPENDIX 2

THE MICROARCHITECTURE OF THE B1700

HERIOT-WATT UNIVERSITY, EDINBURGH

DEPARTMENT OF COMPUTER SCIENCE

A DESCRIPTION OF THE MICROARCHITECTURE OF THE B1700

I. SOMMERVILLE

NOVEMBER, 1976.

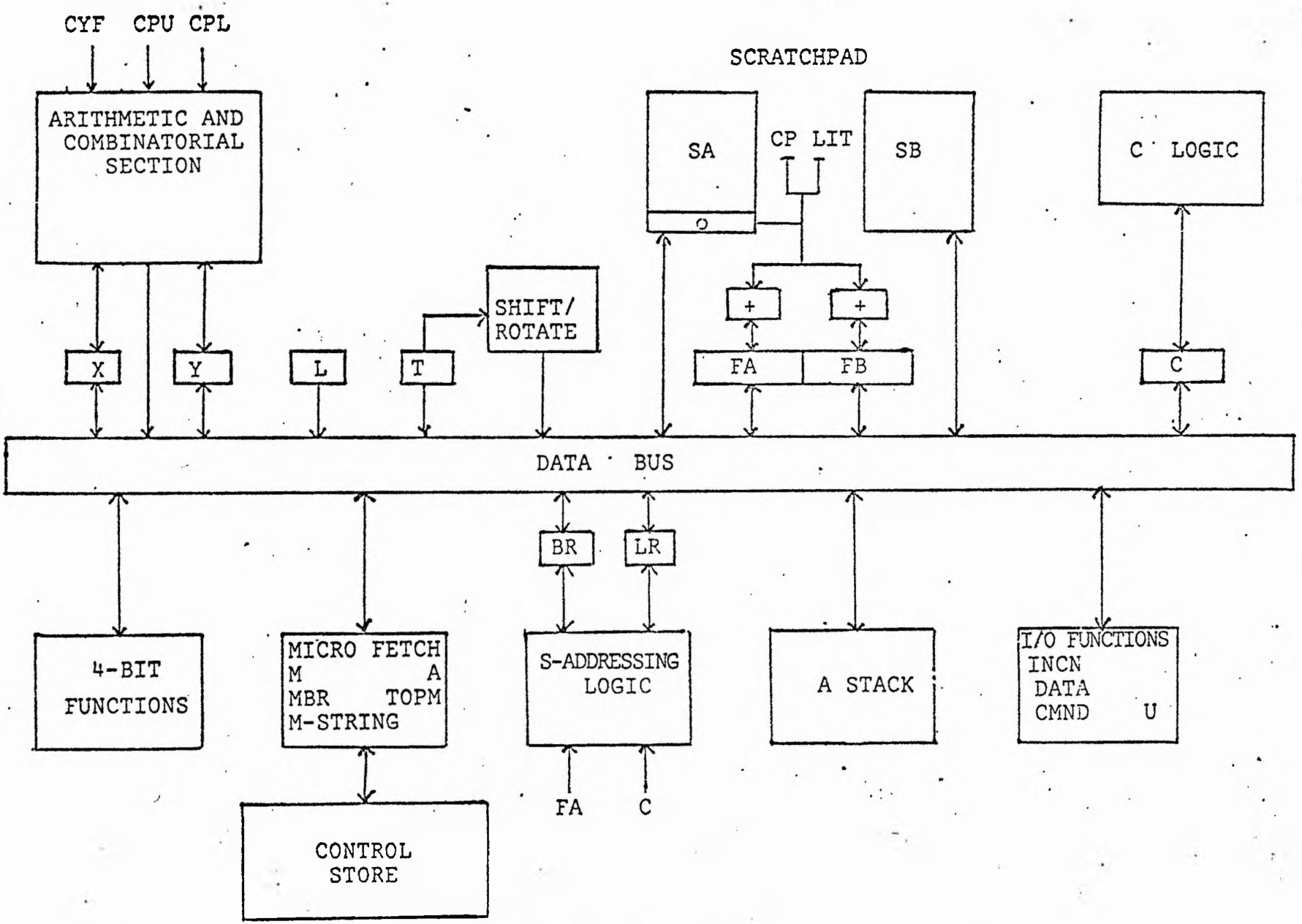
INTRODUCTION

This document describes the micro-architecture of the B1700. Section 1 is a description of the machine registers and their function, section 2 describes how store is addressed, and section 3 covers the B1700 micro-instruction set.

1. THE MICRO-ARCHITECTURE OF THE B1700

The micro-architecture of the B1700 that is, the machine architecture as seen by the microprogrammer is fully documented in the Burroughs B1700 Systems Reference Manual, with a short description of the B1726 processor micro-architecture given below.

The B1726 processor consists of a main store, a high speed microprogram control store and a series of control and combinatorial units and registers, interconnected by one main data bus as shown in figure 1.


CYF
CPU
CPL
ARITHMETIC AND COMBINATORIAL SECTION
SCRATCHPAD
SA
CP
LIT
SB
C LOGIC
SHIFT/ ROTATE
+
+
X
Y
L
T
FA
FB
C
DATA BUS
BR
LR
4-BIT FUNCTIONS
MICRO FETCH
M A
MBR TOPM
M-STRING
S-ADDRESSING LOGIC
A STACK
I/O FUNCTIONS
INCN
DATA
CMND U
FA
C
CONTROL STORE

1.1 THE GENERAL PURPOSE REGISTERS

X, Y, L and T are the general purpose registers for the B1726 processor. They are 24 bits wide and are "active" as data is transferred to and from main memory using these registers. The X and Y registers serve as inputs to the 24 bit function box, T is connected to shift/rotate logic and L in used by the I/O functions. Both L and T may be treated as a group of six 4 bit registers.

1.2 THE FIELD DEFINITION REGISTER

The field definition register (F) specifies the lengths and addresses of data fields in main memory which are to be transferred to or from one of the general purpose registers. The F register is 48 bits wide and is functionally divided into two 24 bit portions (FA and FB). FA specifies the main memory address and FB holds length information.

1.3 THE SCRATCHPAD

This is an array of 16 registers which may be addressed as 16 48-bit words (S0 - S16) or as 32 24-bit words (S0A-S16B). In general, they are used for the storage of frequently accessed information. S0 is treated slightly differently as decision making logic acts on the contents of S0B and FB determining whether S0B is greater than, less than or equal to FB. This can be used for, say, loop termination.

1.4 THE CONTROL REGISTER

The control register (C) is actually a collection of independent registers used by the interrupt system and the combinatorial section of the processor.

1.5 THE INPUT/OUTPUT REGISTERS

The I/O registers (INCN, DATA, CMND, U) are used by the I/O system of the processor for recording interrupts (INCN), loading micro-programs (U) and transferring data to and from the I/O controllers (DATA and CMND).

1.6 THE BASE AND LIMIT REGISTERS

These registers (BR and LR) can be used for main memory protection and for base relative addressing. The processor addressing logic checks if the address in FA falls within their bounds.

1.7 THE A STACK

This is a pushdown store 24 bits wide. It has 32 elements and is designed for use as a micro routine linkage stack. It may also be used for temporary data storage. One appalling feature of its implementation is that it wraps around rather than gives an overflow message when the stack is full.

1.8 THE MICRO REGISTERS

The micro registers (M, A, MBR, TOPM and M-STRING) are registers used in the addressing and execution of micro instructions. A contains the address of the next micro instruction and M the current micro instruction. MBR and TOPM are used to address micro instructions held in main store and M-STRING is used for error diagnosis.

There are other processor registers but these are inherent parts of the control and combinatorial logic and are described along with the control sections.

1.9 THE ARITHMETIC AND COMBINATORIAL SECTION

This section is composed of a 24-bit arithmetic unit and a 24 bit combinatorial unit. It has as data inputs the contents of the X and Y registers as well as the CYF, CPL and CPU sections of the C register. When one of the input registers is changed this section immediately generates a series of results. These results are held in the special purposes register which are those shown in the left hand column in Fig. 2.

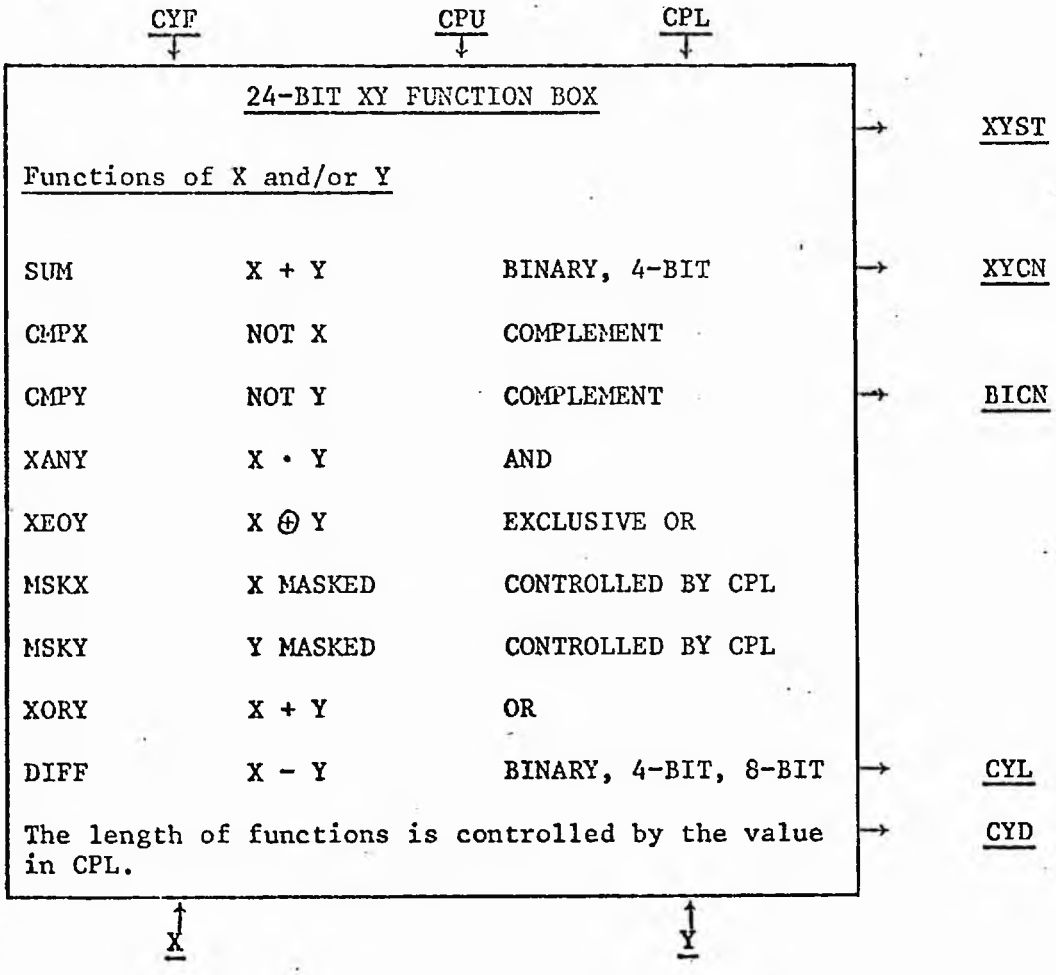


FIG. 2

1.10 SUM REGISTER

This register is the sum of the contents of the X and Y register plus CYF which may hold a carry bit of operands of more than 24 bits are being used. The setting of CPU governs whether the operands are regarded as binary, four-bit decimal or eight-bit decimal.

1.11 DIFF REGISTER

This register holds the difference of the X and Y registers. Again operands may be regarded as binary, four-bit decimal or eight-bit decimal.

1.12 AND/OR/EXCLUSIVE-OR (XANY,XORY,XEOY) REGISTERS

These hold the result of the appropriate logical function AND/OR/XOR of the X and Y registers.

1.13 COMPLEMENT X, COMPLEMENT Y (CMPX, CMPY) REGISTERS

These hold the one's complement of the appropriate register X or Y.

1.14 MASKED X, MASKED Y (MSKX, MSKY) REGISTERS

The mask of the contents of register X or Y is produced and placed in MSKX or MSKY. The value of CPL determines the number of bits masked.

1.15 BINARY CONDITION (BICN) REGISTER

This register holds carry and borrow conditions when operating with quantities greater than 24 bits.

1.16 X/Y CONDITIONS (XYCN) REGISTER

This register holds information on the relative states of X and Y for example X = Y and so on.

1.17 X/Y STATES (XYST) REGISTER

This register holds information on the state of X and Y (are they greater than zero) and also has a bit which is set by any interrupt.

1.18 THE FOUR-BIT FUNCTION BOX

This is an arithmetic and combinatorial function box designed for use with four-bit operands. Its use is governed by the Four-Bit Manipulate microinstruction but has as possible results most of the functions between two operands such as AND/OR/XOR etc.

2. STORE ADDRESSING IN THE B1700

2.1 MICRO-INSTRUCTION ADDRESSING

Micro Instructions for the B1700 are 16 bits in length and are held either in fast control store or in main store. Control store size is either 1024 or 2048 16 bit words. Three registers are used for micro-instruction addressing - A, TOPM and MBR. The addressing mechanism operates as follows:

> The A-register points at the next micro-instruction to be fetched, TOPM points to the top of the control memory in the system and MBR contains the base address in main store above which micro instructions are stored.
>
> When a micro-instruction is to be fetched the value in A is compared with TOPM. If it is less than TOPM, the micro-instruction at address A in the control store is fetched otherwise the micro-instruction at address (A*16) + MBR in main store is fetched. Micro-instructions are held in the M register.

2.2 MAIN-MEMORY ADDRESSING

Main memory is connected to the memory control unit which resolves the addressing conflict between bit oriented data accesses and the physical byte orientation of main memory.

The main store is addressed by a 24 bit absolute address, a 1 bit field direction indicator and a 5-bit field length value which may vary from 0 to 24. The field direction bit indicates whether the memory operation is to be forward or reverse.

The memory control unit (MCU) resolves the bit/byte addressing conflict by fetching the byte addressed by the most significant 21 bits in the address and the three bytes above or below it depending on the field direction indicator. Parity is checked at this stage.

Next the MCU sorts out the actual bits to be transferred from the least signficant 3 bits of the address, the field direction indicator and the field length.

Data is always transferred 24 bits in parallel to and from main memory. Any operation calling for less than 24 bits has leading zeroes supplied by the MCU.

3. The B1700 MICROINSTRUCTION SET

Microinstructions for the B1700 are 16 bits wide and are transferred from control store or main store into the M register. The full instruction set is described below.

REGISTER MOVE

<op code = 1>,<Source Register>,<Destination Register>

This microinstruction is used for arithmetic operations by moving the result from a pseudo operation register.

SCRATCHPAD MOVE

<op code = 2>,<Source or Destination Register>,<D>,<Scratchpad Address>

D indicates direction

0 - to scratchpad
1 - from scratchpad

Scratchpad address indicates the left or rightmost 24 bits of a scratch pad register.

FOUR BIT MANIPULATE

<op code = 3>,<Register to be manipulated>,<Operation>,<Literal>

Operations include skip the next microinstruction if there is a carry or borrow in addition or subtraction.

RELATIVE BRANCH IF BIT TEST FALSE OR TRUE

<op code = 4 or 5>,<Register to be tested>,<B>,<S>,<Literal>

B specifies the bit in the register to be tested. If the S bit is one go to the next microinstruction. Literal is the number to be added to the CSAR if the tested bit meets the condition.

SKIP WHEN

<op code = 6>,<Register to be tested>,<Variant>,<Mask>

Variant specifies condition to be tested for skip to occur. The mask field masks the four bit register selected.

READ OR WRITE MEMORY

<op code = 7>,<RW>,<Variant>,<Register>,<FD>,<Memory field length>

RW indicates read or write. Variant specifies incrementing or decrementing FA and FL. Register is X, Y, T, or L. FD - field direction forward or backward.

MOVE 8 BIT LITERAL

<op code = 8>,<Destination register>,<Eight bit literal>

MOVE 24 BIT LITERAL

<op code = 9>,<Destination register>,<Eight most significant bits of literal>

Sixteen least significant bits of literal are in next control store word.

SHIFT T LEFT

<op code = 10>,<Destination register>,<EC>,<Shift count>

EC - end off or circular.

EXTRACT FROM T REGISTER

<op code = 11>,<Starting bit number>,<Register>,<Number of bits>

Take the specified number of bits starting at the specified bit position and assign them to the destination register: X, Y, T, L.

BRANCH RELATIVE FORWARD OR BACKWARD

<op code = 12 or 13>,<Address>

Branch to the address formed by adding (subtracting) the specified address to (from) the current address.

CALL RELATIVE FORWARD OR BACKWARD

<op code = 14 or 15>,<Address>

Push the next address onto the A stack and then perform a branch as described above.

SWAP MEMORY WITH REGISTER

<op code = 02>,<Register>,<Memory field length>

Swap the specified number of bits in main memory with those in the specified register (X, Y, T, L) in the indicated Field Direction.

CLEAR REGISTER

<op code = 03>,<Register mask>

Set the register indicated to zero. The 8 bits represent the L, T, Y, X, FA, FL, FU, and CP registers.

SHIFT X OR Y

<op code = 04>,<EC>,<LR>,<XY>,<Shift count>

EC - end off or circular
LR - left or right
XY - X or Y register

If shift count is zero, get shift count from CPU register.

SHIFT X AND Y

<op code = 05>,<EC>,<LR>,<Shift count>

Concatenate X and Y and shift.

INCREMENT/DECREMENT FA AND FL

<op code = 06>,<Variant>,<Literal>

Increment and decrement the FA and FL registers by the specified literal according to operation in variant.

EXCHANGE SCRATCHPAD

<op code 07>,<Scratchpad source>,<Scratchpad destination>

Move the F register to the 48 bit scratchpad destination register and move the scratchpad source register to the F register.

INCREMENT/DECREMENT FA REGISTER

<op code = 08>,<ID>,<Scratchpad register>

Increment (or decrement depending on the ID field) the FA register with the contents of the specified scratch pad register.

BIAS

<op code = 003>,<Variant>,<TEST>

Set the CPL register depending on the contents of the FU register specified by the variant. If the test bit is one and CPL is not zero, skip the next microinstruction.

STORE F IN SCRATCHPAD

<op code = 004>,<Scratchpad register>

F is stored in specified scratchpad register.

LOAD F FROM SCRATCHPAD

<op code = 005>,<Scratchpad register>

F is loaded from specified scratchpad register.

SET CARRY FLIP FLOP

<op code = 006>,<Variant>

Set the carry flip flop to one if carry from ALU or borrow from ALU depending on variant.

HALT

<op code = 0001>

Causes machine to halt with the contents of the M register displayed on machine panel.

OVERLAY CONTROL STORE

<op code = 0002>,

Write data from main memory into control store. FA register specifies main memory address. L register specifies control store address. FL register specifies number of bits to transfer.

NORMALIZE

<op code = 0003>

Shift the X register left until the bit specified by the CPL register is one or until the number of bits shifted is the number in the FL register.

APPENDIX 3

EXAMPLES

The material in this appendix is made up of four program listings:-

(1) A listing of the SUILVEN code plus generated microcode for a simple s-machine called SIMPS. This is included to illustrate the format of the output produced by the SUILVEN compiler.

(2) A listing of the SUILVEN code implementing the SASL s-machine.

(3) A listing of the output produced by the B1700 simulator when executing a SASL program to sum the elements of a list.

(4) A listing of the SUILVEN code implementing the PASCAL s-machine.

The code given here and the examples given in the body of the thesis may not exactly correspond. This is due to the fact that both the SASL and the PASCAL machines were re-implemented and opportunity was taken to improve them. The re-implementation was necessary because the author of the thesis left St Andrews University and had no access to the machine there. The programs which were written were supposedly portable but, as usual, this portabilty turned out to be mythical and an inordinate amount of effort was involved in transporting the various programs.

The PASCAL machine implementation was only developed to the stage where the salient features of SUILVEN are illustrated and not to the stage where PASCAL programs actually run on the machine. As we had decided to abandon the development of SUILVEN, we did not feel that the effort of completely implementing the PASCAL machine was justified.

```
        SUILVEN COMPILER -- ST ANDREWS UNIVERSITY-- VERSION 24/8/75                    TODAY IS 29 APRIL 77
        SOURCE LANGUAGE : SNOBOL4(SPITBOL)          TARGET MACHINE : IBM 360/370
        OPTIONS : ON = LIST. OFF = CODE,COPY,DUMP


  1 |      "  THIS IS THE MADILAN CODE FOR SIMPS, A SIMPLE S MACHINE.                  |    1
  1 |          SIMPS IS A STACK BASED MACHINE WITH 6 INSTRUCTIONS.                    |    2
  1 |          LOAD-- 1 PARAMETER WHICH IS AN ADDRESS.LOADS THE CONTENTS OF THAT      |    3
  1 |          ADDRESS ONTO THE STACK.                                                |    4
  1 |          LOADC-- 1 PARAMETER WHICH IS LOADED ONTO THE STACK                     |    5
  1 |          STORE -- 1 PARAMETER WHICH IS AN ADDRESS.STORES THE TOP OF THE STACK   |    6
  1 |          AT THIS ADDRESS                                                        |    7
  1 |          ADD -- ADDS THE TOP 2 STACK ELEMENTS                                   |    8
  1 |          SUB -- SUBTRACTS THE TOP 2 STACK ELEMENTS                              |    9
  1 |          PRINT -- PRINTS THE TOP STACK ELEMENT              "                   |   10
  1 |                                                                                 |   11
COMPILER DIRECTIVE ...CODE
  1 |      BITS(16) ARRAY(200) STACK;       BITS(16) ARRAY (500) STORE;               |   12
                                                          0: CODE BRF(9999)
  3 |          BITS(16) STACK_POINTER,CODEPOINTER,T;                                  |   13
COMPILER DIRECTIVE ...SCRATCH STACK_POINTER,CODEPOINTER,T;
COMPILER DIRECTIVE ...CODE
  4 |                                                                                 |   14
  4 |      PROCEDURE INITIALISE();                                                    |   15
  5 |          CODEPOINTER:=1;     STACK_POINTER:=1;                                  |   16 INITIALISE
                                                          1: CODE X:=1
                                                          2: CODE S15A:=X
                                                          3: CODE X:=1
                                                          4: CODE S15B:=X
  9 |          OUTBUFF_PNT:=1;                                                        |   17 INITIALISE
                                                          5: CODE X:=1
                                                          6: CODE FA:=2084
                                                          7: CODE WRITE(X,24,0)
 10 |          CODEBASE(STORE);                                                       |   18 INITIALISE
 11 |      END;                                                                       |   19 INITIALISE
                                                          8: CODE A:=TAS
```

```
11 |
12 |       FUNCTION GET_PARAMETER();
13 |           T:=STORE(.CODEPOINTER.);

15 |           CODEPOINTER:=CODEPOINTER+1;

16 |       END = T;

16 |
16 |       FUNCTION POPSTACK();
18 |           STACK_POINTER:=STACK_POINTER-1;

20 |       END=STACK(.STACK_POINTER.);

20 |
```

```
                                        |    20
                                        |    21
                                        |    22 GET_PARAMETER
 9: CODE X:=S15A
10: CODE Y:=16
11: CODE X:=X*Y
12: CODE Y:=6304
13: CODE FA:=SUM
14: CODE READ(X,16,0)
15: CODE S14B:=X
                                        |    23 GET_PARAMETER
16: CODE X:=S15A
17: CODE Y:=1
18: CODE X:=SUM
19: CODE S15A:=X
                                        |    24 GET_PARAMETER
20: CODE X:=S14B
21: CODE A:=TAS
                                        |    25
                                        |    26
                                        |    27 POPSTACK
22: CODE X:=S15B

23: CODE Y:=1
24: CODE X:=DIFF
25: CODE S15B:=X
                                        |    28 POPSTACK
26: CODE X:=S15B
27: CODE Y:=16
28: CODE X:=X*Y
29: CODE Y:=3104
30: CODE FA:=SUM
31: CODE READ(X,16,0)
32: CODE A:=TAS
                                        |    29
```

```
 20 |      PROCEDURE PUSHSTACK(BITS(16) P;);
COMPILER DIRECTIVE ...SCRATCHCOPY P

 22 |           STACK(.STACK_POINTER.):=P;

 24 |           STACK_POINTER:=STACK_POINTER+1;

 25 |           IF STACK_POINTER > 200 THEN

 26 |                $(
 27 |                     WRITESTRING('     STACK OVERFLOW -
```

```
                                        |    30

   33: CODE FA:=14368
   34: CODE READ(X,16,0)
   35: CODE S14A:=X
                                        |    31 PUSHSTACK
   36: CODE X:=S15B
   37: CODE Y:=16
   38: CODE X:=X*Y
   39: CODE Y:=3104
   40: CODE TAS:=SUM
   41: CODE X:=S14A
   42: CODE FA:=TAS
   43: CODE WRITE(X,16,0)
                                        |    32 PUSHSTACK
   44: CODE X:=S15B
   45: CODE Y:=1
   46: CODE X:=SUM
   47: CODE S15B:=X
                                        |    33 PUSHSTACK
   48: CODE X:=S15B
   49: CODE TAS:=X
   50: CODE X:=200
   51: CODE Y:=X
   52: CODE X:=TAS
   53: CODE BTBT(XYCN,0,1)
                                        |    34 PUSHSTACK
- JOB ZAPPED ')  ;                      |    35 PUSHSTACK
   55: CODE X:=S1B
   56: CODE Y:=33
   57: CODE S1B:=SUM
   58: CODE SHIFT(X,L3)
   59: CODE Y:=1080
   60: CODE FA:=SUM
   61: CODE X:=H404040
   62: CODE WRITE(X,24,1)
   63: CODE X:=H40E2E3
   64: CODE WRITE(X,24,1)
```

```
23 |                     WRITE();
29 |                 $)
30 |     END;
30 |
30 |     PROCEDURE LOAD();
```

```
 65: CODE X:=HC1C3D2
 66: CODE WRITE(X,24,1)
 67: CODE X:=H40D6E5
 68: CODE WRITE(X,24,1)
 69: CODE X:=HC5D9C6
 70: CODE WRITE(X,24,1)
 71: CODE X:=HD3D6E6
 72: CODE WRITE(X,24,1)
 73: CODE X:=H406D6D
 74: CODE WRITE(X,24,1)
 75: CODE X:=H40D1D6
 76: CODE WRITE(X,24,1)
 77: CODE X:=HC240E9
 78: CODE WRITE(X,24,1)
 79: CODE X:=HC1D7D7
 80: CODE WRITE(X,24,1)
 81: CODE X:=HC5C440
 82: CODE WRITE(X,24,1)
                                  |    36 PUSHSTACK
 83: CODE X:=1080
 84: CODE L:=120
 85: CODE Y:=1
 86: CODE S1B:=Y
 87: CODE WRITE
 88: CODE X:=H404040
 89: CODE FA:=1080
 90: CODE FL:=960
 91: CODE WRITE(X,24,3)
 92: CODE BTBT(FLCN,0,-2)
                                  |    37 PUSHSTACK
                                  |    38 PUSHSTACK
***** CODE JUMP:=+38 AT 54
 93: CODE A:=TAS
                                  |    39
```

```
 31 |      PROCEDURE LOADC();
 32 |          PUSHSTACK(STORE(.GET_PARAMETER().));


 34 |      END;

 34 |
 34 |      PROCEDURE LOADC();
 36 |      BITS(16) TEMP;
COMPILER DIRECTIVE ...SCRATCH TEMP
 37 |      TEMP:=GET_PARAMETER();


 39 |      PUSHSTACK(TEMP);



 40 |      END;
```

```
                                        |    40
                                        |    41 LOAD
  94: CODE CALLR(86)
  95: CODE Y:=16
  96: CODE X:=X*Y
  97: CODE Y:=6304
  98: CODE FA:=SUM
  99: CODE READ(X,16,0)
 100: CODE FA:=14368
 101: CODE WRITE(X,16,0)
 102: CODE CALLR(70)
                                        |    42 LOAD
 103: CODE A:=TAS
                                        |    43
                                        |    44
                                        |    45 LOADC

                                        |    46 LOADC
 104: CODE CALLR(96)
 105: CODE S14A:=X
                                        |    47 LOADC
 106: CODE X:=S14A
 107: CODE FA:=14368
 108: CODE WRITE(X,16,0)
 109: CODE CALLR(77)
                                        |    48 LOADC
```

```
40 |
41 |       PROCEDURE STORE();
42 |           STORE(.GET_PARAMETER().):=POPSTACK();
43 |
44 |       END;
45 |
46 |
47 |       PROCEDURE ADD();
48 |           T:=POPSTACK() + POPSTACK();
49 |
50 |           PUSHSTACK(T);
51 |
52 |       END;
53 |
54 |       PROCEDURE SUB();
55 |           PUSHSTACK(POPSTACK() - POPSTACK());
56 |
57 |       END;
```

```
110: CODE A:=TAS
                                    |   49
                                    |   50
                                    |   51 STORE
111: CODE CALLR(103)
112: CODE Y:=16
113: CODE X:=X*Y
114: CODE Y:=6304
115: CODE TAS:=SUM
116: CODE CALLR(95)
117: CODE FA:=TAS
118: CODE WRITE(X,16,0)
                                    |   52 STORE
119: CODE A:=TAS
                                    |   53
                                    |   54
                                    |   55 ADD
120: CODE CALLR(99)
121: CODE TAS:=X
122: CODE CALLR(101)
123: CODE Y:=X
124: CODE X:=TAS
125: CODE X:=SUM
126: CODE S14B:=X
                                    |   56 ADD
127: CODE X:=S14B
128: CODE FA:=14368
129: CODE WRITE(X,16,0)
130: CODE CALLR(98)
                                    |   57 ADD
131: CODE A:=TAS
                                    |   58
                                    |   59
                                    |   60 SUB
132: CODE CALLR(111)
133: CODE TAS:=X
134: CODE CALLR(113)
135: CODE Y:=X
136: CODE X:=TAS
137: CODE X:=DIFF
138: CODE FA:=14368
139: CODE WRITE(X,16,0)
140: CODE CALLR(108)
```

```
52 |
53 |       PROCEDURE PRINT();
55 |           WRITESTRING('          THE RESULT IS
```

```
                                        |    63
                                        |    64 PRINT
*);
142: CODE X:=S1B
143: CODE Y:=21
144: CODE S1B:=SUM
145: CODE SHIFT(X,L3)
146: CODE Y:=1080
147: CODE FA:=SUM
148: CODE X:=H404040
149: CODE WRITE(X,24,1)
150: CODE X:=H404040
151: CODE WRITE(X,24,1)
152: CODE X:=H40E3C8
153: CODE WRITE(X,24,1)
```

```
154: CODE X:=HC540D9
155: CODE WRITE(X,24,1)
156: CODE X:=HC5E2E4
157: CODE WRITE(X,24,1)
158: CODE X:=HD3E340
159: CODE WRITE(X,24,1)
160: CODE X:=HC9E240
161: CODE WRITE(X,24,1)
57 |       WRITENUMBER(STACK(.STACK_POINTER-1.));                    |   65 PRINT
162: CODE X:=S15B
163: CODE Y:=1
164: CODE X:=DIFF
165: CODE Y:=16
166: CODE X:=X*Y
167: CODE Y:=3104
168: CODE FA:=SUM
169: CODE READ(X,16,0)
170: CODE FA:=352
171: CODE FL:=80
172: CODE Y:=10
173: CODE X/Y
174: CODE TAS:=X
175: CODE X:=240
176: CODE X:=SUM
177: CODE WRITE(X,8,3)
178: CODE X:=TAS
179: CODE BTBT(XYST,0,-8)
180: CODE CFAFL(FA,1,5)
181: CODE X:=S1B
182: CODE SHIFT(X,L3)
183: CODE Y:=1080
184: CODE S2A:=SUM
185: CODE TAS:=80
186: CODE S0B:=TAS
187: CODE READ(X,8,4)
188: CODE XCHF(2,2)
189: CODE WRITE(X,8,1)
190: CODE XCHF(2,2)
191: CODE BTBT(FLCN,1,-5)
```

```
 58 |          WRITE();
 59 |      END;
 59 |
 59 |      PROCEDURE MAINLOOP();
 61 |      BITS(16) OPCODE;
COMPILER DIRECTIVE ...SCRATCH OPCODE
 62 |          OPCODE:=GET_PARAMETER();

 64 |          WHILE OPCODE ^= 15 DO

 65 |            $(
 66 |              CASE OPCODE OF
```

```
191: CODE BTBT(FLCN,0,-2)                    |   66 PRINT
192: CODE X:=1080
193: CODE L:=120
194: CODE Y:=1
195: CODE S1B:=Y
196: CODE WRITE
197: CODE X:=H404040
198: CODE FA:=1080
199: CODE FL:=960
200: CODE WRITE(X,24,3)
201: CODE BTBT(FLCN,0,-2)
                                             |   67 PRINT
202: CODE A:=TAS
                                             |   68
                                             |   69
                                             |   70 MAINLOOP
                                             |   71 MAINLOOP
203: CODE CALLR(195)
204: CODE S14A:=X
                                             |   72 MAINLOOP
205: CODE X:=S14A
206: CODE TAS:=X
207: CODE X:=15
208: CODE Y:=X
209: CODE X:=TAS
210: CODE BTBF(XYCN,2,1)
                                             |   73 MAINLOOP
                                             |   74 MAINLOOP
212: CODE X:=S14A
213: CODE TAS:=X
```

```
67 |                LOAD();                                                  |  75 MAINLOOP
                                                  215: CODE CALLR(122)
68 |                STORE();                                                 |  76 MAINLOOP
                                                  217: CODE CALLR(107)
69 |                ADD();                                                   |  77 MAINLOOP
                                                  219: CODE CALLR(100)
70 |                SUB();                                                   |  78 MAINLOOP
                                                  221: CODE CALLR(90)
71 |                PRINT();                                                 |  79 MAINLOOP
                                                  223: CODE CALLR(82)
72 |                LOADC();                                                 |  80 MAINLOOP
                                                  225: CODE CALLR(122)
73 |             ENDCASE;                                                    |  81 MAINLOOP
                                                ****: CODE JUMP:=+12 AT 214
                                                  227: CODE M:=TAS
                                                  228: CODE BRF(0)
                                                  229: CODE NOP
                                                  230: CODE BRB(16)
                                                  231: CODE BRB(15)
                                                  232: CODE BRB(14)
                                                  233: CODE BRB(13)
                                                  234: CODE BRB(12)
                                                  235: CODE BRB(11)
                                                ****: CODE JUMP:=+19 AT 216
                                                ****: CODE JUMP:=+17 AT 218
                                                ****: CODE JUMP:=+15 AT 220
                                                ****: CODE JUMP:=+13 AT 222
                                                ****: CODE JUMP:=+11 AT 224
                                                ****: CODE JUMP:=+9 AT 226
73 |          OPCODE:=GET_PARAMETER();                                       |  82 MAINLOOP
                                                  236: CODE CALLR(228)
                                                  237: CODE S14A:=X
74 |             $)                                                          |  83 MAINLOOP
75 |       END:                                                              |  84 MAINLOOP
```

```
                                                    238: CODE BRB(34)
                                                  *****: CODE JUMP:=+27 AT 211
                                                    239: CODE A:=TAS
 75 |                                                                          |  85
 75 |     "    THIS IS THE MAIN PROGRAM  "                                    |  86
 75 |                                                                          |  87


 75 |                INITIALISE();                                             |  88
                                                    240: CODE CALLR(240)
 77 |                MAINLOOP();                                               |  89
                                                    241: CODE CALLR(39)
 79 |                                                                          |  90
 79 | ENDPROGRAM                                                               |  91
                                                    242: CODE HALT
```

****

IMPILATION COMPLETE -- NO OF ERRORS = 0

```
1 |  "*************************************************************************  |   1
1 |                                                                              |   2
1 |    THIS IS THE SUILVEN CODE FOR THE SASL S_MACHINE WHICH WAS DESIGNED        |   3
1 |  AT ST ANDREWS UNIVERSITY FOR THE LIST PROCESSING LANGUAGE SASL.             |   4
1 |  THE SASL MACHINE HAS A TAGGED ARCHITECTURE ,WITH TYPE                       |   5
1 |  CHECKING CARRIED OUT AT RUN TIME                                            |   6
1 |                                                                              |   7
1 |    PROGRAMMER : IAN SOMMERVILLE -- ST ANDREWS UNIVERSITY -- 23/8/75          |   8
1 |                                                                              |   9
1 |         IN ADDITION TO THE MACHINE STACK , THE SASL MACHINE HAS A            |  10
1 |         LIST STORAGE AREA , USED FOR BOTH PROGRAM AND DATA , AND TWO         |  11
1 |         SPECIAL PURPOSE REGISTERS CALLED CREG AND EREG . CREG POINTS TO      |  12
1 |         THE NEXT MACHINE INSTRUCTION TO BE EXECUTED( OR TO THE PARAMETER     |  13
1 |               OF THE CURRENT INSTRUCTIONIF IT HAS ONE) AND EREG POINTS TO    |  14
1 |         THE CURRENT PROGRAM ENVIRONMENT .                                    |  15
1 |         THERE ARE THREE TYPES OF MACHINE INSTRUCTIONS                        |  16
1 |         1) THOSE WHICH TAKE TWO OPERANDS FROM THE STACK AND PUT ONE          |  17
1 |            OPERAND BACK ON THE STACK                                         |  18
1 |         2)  THOSE WHICH TAKE ONE OPERAND FROM THE STACK AND PUT ONE BACK     |  19
1 |               ONTO THE STACK                                                 |  20
1 |         3) THOSE WHICH HAVE A PARAMETER . THESE MAY OR MAY NOT ALSO          |  21
1 |            MANIPULATE THE STACK                                              |  22
1 |                                                                              |  23
1 |                                                                              |  24
```

```
1 |                                                                                                                    |  25
1 |       THE SASL OP CODES FOLLOW                                                                                     |  26
1 |                                                                                                                    |  27
1 |                                                                                                                    |  28
1 |                                                                                                                    |  29
1 |       OP CODE = 1        DIV     INTEGER DIVIDE                                                                    |  30
1 |       OP CODE = 2        MOD     MODULUS                                                                           |  31
1 |       OP CODE = 3        PLUS    INTEGER ADDITION                                                                  |  32
1 |       OP CODE = 4        MINUS   INTEGER SUBTRACTION                                                               |  33
1 |       OP CODE = 5        MULT    INTEGER MULTIPLICATION                                                            |  34

1 |       OP CODE = 6        GRT     TEST GREATER THAN                                                                 |  35
1 |       OP CODE = 7        GEQ     TEST GREATER THAN OR EQUALS                                                       |  36
1 |       OP CODE = 8        LTH     TEST LESS THAN                                                                    |  37
1 |       OP CODE = 9        LEQ     TEST LESS THAN OR EQUALS                                                          |  38
1 |       OP CODE = 10       AND      LOGICAL AND                                                                      |  39
1 |       OP CODE = 11       OR      LOGICAL OR                                                                        |  40
1 |       OP CODE = 12       COMMA   ADDS ELEMENT TO A LIST                                                            |  41
1 |       OP CODE = 13       APPLY   APPLIES A SASL FUNCTION                                                           |  42
1 |       OP CODE = 14       HEAD    GETS HEAD OF LIST                                                                 |  43
1 |       OP CODE = 15       TAIL    GETS TAIL OF LIST                                                                 |  44
1 |       OP CODE = 16       TLG     TEST LOGICAL TYPE                                                                 |  45
1 |       OP CODE = 17       TCH     TEST CHARACTER TYPE                                                               |  46
1 |       OP CODE = 18       TP      TEST POINTER TYPE                                                                 |  47
1 |       OP CODE = 19       TF      TEST FUNCTION TYPE                                                                |  48
1 |       OP CODE = 20       TDIG    TEST DIGIT                                                                        |  49
1 |       OP CODE = 21       TLET    TEST LETTER                                                                       |  50
1 |       OP CODE = 22       DIGVAL  GET DIGIT(IN CHAR FORM ) VALUE                                                    |  51
1 |       OP CODE = 23       NEG     NEGATIVE TOP OF STACK                                                             |  52
1 |       OP CODE = 24       POSITIVE TOP OF STACK                                                                     |  53
1 |       OP CODE = 25       NOTTOP  LOGICAL NOT                                                                       |  54
1 |       OP CODE = 26       FORK    CODE BRANCH                                                                       |  55
1 |       OP CODE = 27       TEQ     TESTS FOR EQUALITY                                                                |  56
1 |       OP CODE = 28       NEQ     TESTS FOR INEQUALITY                                                              |  57
1 |       OP CODE = 29       DECL    PROCESSES SASL DECLARATIONS(NON RECURSIVE)                                        |  58
1 |       OP CODE = 30       DECG    PROCESSES RECURSIVE SASL DECLARATIONS                                             |  59
1 |       OP CODE = 31       KNOT    FILLS IN ENVIRONMENT LIST                                                         |  60
1 |       OP CODE = 32       BLOCK   BLOCK ENTRY                                                                       |  61
1 |                                                                                                                    |  62
1 |  ***************************************************************************"                                    |  63
```

```
1  |       MACRO 'INTEGER_TYPE' = '1';                MACRO 'LOGICAL_TYPE' = '2';
3  |       MACRO 'CHAR_TYPE' = '3';                 MACRO 'GUESS_TYPE' = '4';
5  |       MACRO 'FUNCTION_TYPE' = '5';             MACRO 'POINTER_TYPE' = '6';
7  |
7  |       MACRO 'LTRUE' = '1';                       MACRO 'LFALSE' = '0';
9  |
9  |       MACRO 'NIL' = '0';
10 |
10 |       MACRO 'STACK_SIZE' = '250';                MACRO 'STACK_WIDTH' = '24';
12 |       MACRO 'LIST_SIZE' = '3000';                MACRO 'LIST_WIDTH' = '45';
14 |       MACRO 'TAG_SIZE' = '4';                    MACRO 'WORD_SIZE' = '20';
15 |
15 |
16 |     MACRO 'SAVE_VALUES_OF_POINTERS' = ' STACK(.STACK_POINTER.).DATA:=PL1;
17 |                                        STACK(.STACK_POINTER+1.).DATA:=PL2;
17 |                                        STACK_POINTER:=STACK_POINTER+2;  ';
17 |
17 |      MACRO 'RESTORE_VALUES_OF_POINTERS' =
18 |                                      ' STACK_POINTER:=STACK_POINTER-2;
18 |                                        PL1:=STACK(.STACK_POINTER.).DATA;
18 |                                        PL2:=STACK(.STACK_POINTER+1.).DATA;  ';
18 |
18 |
18 |
18 |  MACRO 'SAVE_MACHINE_STATE' = ' SECS_TAG:=POINTER_TYPE;
19 |         SECS_DATA:=CREG;   STACK(.STACK_POINTER.).TAG:=POINTER_TYPE;
19 |         STACK(.STACK_POINTER.).DATA:=EREG;
19 |         STACK_POINTER:=STACK_POINTER+1;      ';
19 |
19 |  MACRO 'RESTORE_MACHINE_STATE' = 'STACK_POINTER:=STACK_POINTER-1;
20 |          EREG:=STACK(.STACK_POINTER.).DATA;
20 |          CREG:=SECS_DATA;
20 |          SECS_TAG:=TOPS_TAG;
20 |          SECS_DATA:=TOPS_DATA;
20 |          UNSET(TOPSF);
20 |          ' ;
20 |
20 |
```

COMPILER DIRECTIVE ...PAGE

```
20 |       "  DEFINE THE MACHINE ARCHITECTURE USING BITS DECLARATIONS      "                    | 103
20 |                                                                                            | 104
20 |       BITS(STACK_WIDTH) ARRAY (STACK_SIZE) STACK;                                          | 105
21 |       BITS(LIST_WIDTH) ARRAY (LIST_SIZE) LIST;                                             | 106
22 |       BITS(WORD_SIZE) EREG,CREG;                                                           | 107
23 |                                                                                            | 108
23 |       "     NOW DEFINE SOME OTHER STORAGE AREAS USED BY THE SASL MACHINE                   | 109
23 |                BUT NOT VISIBLE TO THE SASL PROGRAM                                         | 110
23 |             THESE DATA AREAS ARE:                                                          | 111
23 |                   FREE_SPACE_LIST USED TO RECORD WHICH LIST CELLS ARE                      | 112
23 |                      AVAILABLE FOR USE                                                     | 113
23 |             STACK_POINTER     OBVIOUS                                                      | 114
23 |             OP_CODE      THE CURRENT INSTRUCTION OP_CODE                                   | 115
23 |             LIST_POINTER      POINTER INTO LIST AREA                                       | 116
23 |             FSLP     POINTER INTO FREE SPACE LIST                                         | 117
23 |             TOPS_TAG,TOPS_DATA,SECS_TAG,SECS_DATA  THESE ARE FILLED BY THE                | 118
23 |                POPSTACK INSTRUCTIONS AND ARE USED WHEN WE WISH TO OPERATE                  | 119
23 |                ON THE TOP STACK ELEMENTS                                                   | 120
23 |             INTERRUPT_TYPE HOLDS THE TYPE OF INTERRUPT WHICH HAS OCCURRED                 | 121
23 |       "                                                                                    | 122
23 |                                                                                            | 123
23 |       BITS(WORD_SIZE) ARRAY ( LIST_SIZE) FREE_SPACE_LIST;                                  | 124
24 |       BITS(WORD_SIZE) STACK_POINTER,OP_CODE,LIST_POINTER,FSLP,TOPS_DATA,SECS_DATA          | 125
25 |             ,TOPS_TAG,SECS_TAG,INTERRUPT_TYPE,RECLEVEL;                                   | 126
25 |                                                                                            | 127
25 |                                                                                            | 128
COMPILER DIRECTIVE ...PAGE
```

```
 25 |       "     DEFINE THE STACK AND LIST AREA STRUCTURES USING TEMPLATE                  |  129
 25 |          DECLARATIONS AND ASSIGN THESE TO STACK AND LIST                              |  130
 25 |       "                                                                               |  131
 25 |       TEMPLATE STACK_STRUCTURE = TAG(TAG_SIZE),DATA(WORD_SIZE);                       |  132
 26 |       DEFINE STACK_STRUCTURE: STACK;                                                  |  133
 27 |                                                                                       |  134
 27 |       TEMPLATE LIST_STRUCTURE = GC(1), " A BIT USED BY THE GARBAGE COLLECTOR"         |  135
 28 |                                 TAG(TAG_SIZE),HEAD(WORD_SIZE),TAIL(WORD_SIZE);        |  136
 28 |       DEFINE LIST_STRUCTURE : LIST ;                                                  |  137
 29 |                                                                                       |  138
 29 |                                                                                       |  139
COMPILER DIRECTIVE ...SCRATCH CREG,EREG,STACK_POINTER,LIST_POINTER,FSLP,OP_CODE
COMPILER DIRECTIVE ...SCRATCH TOPS_TAG,TOPS_DATA,SECS_TAG,SECS_DATA
 29 |                                                                                       |  140
 29 |       "     DEFINE THE MACHINE INTERRUPT FLAGS        "                           |  141
 29 |                                                                                       |  142
 29 |       FLAG STACK_OVERFLOW,INVALID_TYPE,DIVZERO,FUNCTION_COMPARISON,GUESS_FLAG,        |  143
 30 |                  FIRST_ENTRY,ENDPROG;                                                 |  144
 30 |     FLAG                                                                              |  145
 31 |         DECLFLAG,                                                                     |  146
 31 |         SECSF,                                                                        |  147
 31 |         IOPSF;                                                                        |  148
 31 |                                                                                       |  149
 31 |                                                                                       |  150
COMPILER DIRECTIVE ...PAGE
```

```
31 |     PROCEDURE HANDLE_INTERRUPTS();
33 |        WRITESTRING('INTERRUPT ');   WRITENUMBER(INTERRUPT_TYPE);
36 |        WRITESTRING('          JOB ZAPPED **********');
37 |        WRITE();
38 |         HALT;
39 |      END;
39 |
39 |
39 | PROCEDURE FILL_TOPSTACK_REGISTERS();
41 |    IF FALSE(TOPSF) THEN
43 |      $(
44 |         IF FALSE(SECSF) THEN
45 |             $(
46 |                TOPS_TAG:=STACK(.STACK_POINTER.).TAG;
47 |                TOPS_DATA:= STACK(.STACK_POINTER.).DATA;
48 |                STACK_POINTER:=STACK_POINTER-1;
49 |             $)
50 |         ELSE
50 |             $(
51 |                TOPS_TAG:=SECS_TAG;
52 |                TOPS_DATA:=SECS_DATA;
53 |             $)
54 |         SECS_TAG:=STACK(.STACK_POINTER.).TAG;
55 |         SECS_DATA:=STACK(.STACK_POINTER.).DATA;
56 |         SET(TOPSF);       SET(SECSF);
58 |         STACK_POINTER:=STACK_POINTER-1;
59 |      $)
60 | END;
60 |
60 |
60 |
60 |      PROCEDURE INITIALISE_SASL_MACHINE();
62 |         STACK_POINTER:=0;                LIST_POINTER:=1;
65 |         UNSET(SECSF);    UNSET(TOPSF);
67 |         CREG:=1;                EREG:=1;
69 |         CODEBASE(LIST);
70 |      END;
70 |
70 |
70 |
```

```
151 |
152 | HANDLE_INTERRUPTS
153 | HANDLE_INTERRUPTS
154 | HANDLE_INTERRUPTS
155 | HANDLE_INTERRUPTS
156 | HANDLE_INTERRUPTS
157 |
158 |
159 |
160 | FILL_TOPSTACK_REGISTER
161 | FILL_TOPSTACK_REGISTER
162 | FILL_TOPSTACK_REGISTER
163 | FILL_TOPSTACK_REGISTER
164 | FILL_TOPSTACK_REGISTER
165 | FILL_TOPSTACK_REGISTER
166 | FILL_TOPSTACK_REGISTER
167 | FILL_TOPSTACK_REGISTER
168 | FILL_TOPSTACK_REGISTER
169 | FILL_TOPSTACK_REGISTER
170 | FILL_TOPSTACK_REGISTER
171 | FILL_TOPSTACK_REGISTER
172 | FILL_TOPSTACK_REGISTER
173 | FILL_TOPSTACK_REGISTER
174 | FILL_TOPSTACK_REGISTER
175 | FILL_TOPSTACK_REGISTER
176 | FILL_TOPSTACK_REGISTER
177 | FILL_TOPSTACK_REGISTER
178 | FILL_TOPSTACK_REGISTER
179 |
180 |
181 |
182 |
183 | INITIALISE_SASL_MACHIN
184 | INITIALISE_SASL_MACHIN
185 | INITIALISE_SASL_MACHIN
186 | INITIALISE_SASL_MACHIN
187 | INITIALISE_SASL_MACHIN
188 |
189 |
190 |
```

```
 70 |                                                                                                  |  191
 70 |  PROCEDURE PUSHSTACK(BITS(TAG_SIZE) TAG; BITS(WORD_SIZE) DATA;);                                 |  192
 72 |                                                                                                  |  193 PUSHSTACK
 72 |       "   PUSHES AN ELEMENT ONTO THE SASL MACHINE STACK   "                                     |  194 PUSHSTACK
 72 |                                                                                                  |  195 PUSHSTACK
 72 |      IF FALSE(TOPSF) THEN                                                                        |  196 PUSHSTACK
 74 |       $(                                                                                         |  197 PUSHSTACK
 75 |          IF FALSE(SECSF) THEN                                                                    |  198 PUSHSTACK
 76 |            $(                                                                                    |  199 PUSHSTACK
 77 |               SECS_TAG:=TAG;     SET(SECSF);                                                     |  200 PUSHSTACK
 79 |               SECS_DATA:=DATA;                                                                   |  201 PUSHSTACK
 80 |            $)                                                                                    |  202 PUSHSTACK
 81 |          ELSE                                                                                    |  203 PUSHSTACK
 81 |            $(                                                                                    |  204 PUSHSTACK
 82 |               TOPS_TAG:=TAG;     SET(TOPSF);                                                     |  205 PUSHSTACK
 84 |               TOPS_DATA:=)ATA;                                                                   |  206 PUSHSTACK
 85 |            $)                                                                                    |  207 PUSHSTACK
 86 |     $)                                                                                           |  208 PUSHSTACK
 87 |    ELSE                                                                                          |  209 PUSHSTACK
 87 |            IF STACK_POINTER > STACK_SIZE THEN                                                    |  210 PUSHSTACK
 88 |              $(                                                                                  |  211 PUSHSTACK
 89 |                 SET(ANY_INTERRUPT);                                                              |  212 PUSHSTACK
 90 |                 INTERRUPT_TYPE:=1;                                                               |  213 PUSHSTACK
 91 |              $)                                                                                  |  214 PUSHSTACK
 92 |            ELSE                                                                                  |  215 PUSHSTACK
 92 |              $(                                                                                  |  216 PUSHSTACK
 93 |                 STACK(.STACK_POINTER.):=SECS_TAG SHL WORD_SIZE | DATA;                           |  217 PUSHSTACK
 94 |                 SECS_TAG:=TOPS_TAG;  SECS_DATA:=TOPS_DATA;                                       |  218 PUSHSTACK
 96 |                 TOPS_TAG:=TAG;      TOPS_DATA:=DATA;                                             |  219 PUSHSTACK
 93 |                 STACK_POINTER:=STACK_POINTER + 1;                                                |  220 PUSHSTACK
 99 |              $)                                                                                  |  221 PUSHSTACK
100 |  END;    "  PUSHSTACK  "                                                                        |  222 PUSHSTACK
100 |                                                                                                  |  223
100 |                                                                                                  |  224
.. COMPILER DIRECTIVE ...PAGE
```

```
100 |       "  HERE ARE SOME UTILITY PROCEDURES          "                                  |  225
109 |                                                                                       |  226
109 |       "                  THE GARBAGE COLLECTION PROCEDURES                            |  227
109 |          GARBAGE COLLECTION IS DONE WHEN THE FREE SPACE LIST IS EMPTY                 |  228
109 |          WHEN THIS OCCURS THE GARBAGE COLLECTOR IS AUTOMATICALLY CALLED               |  229
109 |          THE METHOD USED IS TO CHAIN THROUGH THE LIST AREA USING CREG , EREG ,        |  230
109 |          AND ALL THE POINTER TYPE ELEMENTS ON THE STACK .                             |  231
100 |       EACH CELL WHICH IS ACCESSIBLE(POINTED TO) IS TAGGED 1(THE GC BIT)               |  232
109 |          WHEN THIS MARKING IS COMPLETE WE SCAN THROUGH THE WHOLE LIST AREA            |  233
109 |          COLLECTING UNMARKED CELLS AND ADDING THEM TO THE FREE SPACE LIST             |  234
109 |       "                                                                               |  235
100 |       PROCEDURE MARK_LIST(BITS(WORD_SIZE) R;);                                        |  236
102 |                                                                                       |  237 MARK_LIST
102 |       "      DOES THE LIST MARKING       "                                            |  238 MARK_LIST
102 |                                                                                       |  239 MARK_LIST
102 |          REPEAT                                                                       |  240 MARK_LIST
104 |             LIST(.R.).GC:=1                                                           |  241 MARK_LIST
105 |          UNTIL R=NIL DO                                                               |  242 MARK_LIST
105 |             R:=LIST(.R.).TAIL;                                                        |  243 MARK_LIST
106 |       END;                                                                            |  244 MARK_LIST
106 |                                                                                       |  245
106 |                                                                                       |  246
105 |       PROCEDURE GARBAGE_COLLECT();                                                    |  247
108 |                                                                                       |  248 GARBAGE_COLLECT
108 |          BITS(WORD_SIZE) TP;                                                          |  249 GARBAGE_COLLECT
COMPILER DIRECTIVE ...SCRATCH TP
109 |          MARK_LIST(CREG);                          MARK_LIST(EREG);                   |  250 GARBAGE_COLLECT
112 |          TP:=1;                                                                       |  251 GARBAGE_COLLECT
113 |          WHILE TP<=STACK_POINTER DO                                                   |  252 GARBAGE_COLLECT
114 |             IF STACK(.TP.).TAG=POINTER_TYPE THEN                                      |  253 GARBAGE_COLLECT
115 |                MARK_LIST(STACK(.TP.).DATA);                                           |  254 GARBAGE_COLLECT
116 |          TP:=1;                        FSLP:=1;                                       |  255 GARBAGE_COLLECT
113 |          WHILE TP<=LIST_SIZE DO                                                       |  256 GARBAGE_COLLECT
112 |             IF LIST(.TP.).GC=0 THEN                                                   |  257 GARBAGE_COLLECT
120 |                $(                                                                     |  258 GARBAGE_COLLECT
121 |                   FREE_SPACE_LIST(.FSLP.):=TP;          FSLP:=FSLP+1;                |  259 GARBAGE_COLLECT
123 |                $)                                                                     |  260 GARBAGE_COLLECT
124 |             ELSE                                                                      |  261 GARBAGE_COLLECT
124 |                LIST(.TP.).GC:=0;                                                      |  262 GARBAGE_COLLECT
125 |       END;                                                                            |  263 GARBAGE_COLLECT
                                                                                            |  264
```

```
125 |       FUNCTION GET_NEW_CELL();                                                                  |  265
127 |                                                                                                 |  266 GET_NEW_CELL
127 |       "  THIS FUNCTION RETURNS THE ADDRESS OF A FREE LIST CELL WHEN CALLED FOR                  |  267 GET_NEW_CELL
127 |          IF THERE ARE NONE AVAILABLE IN THE FREE SPACE LIST IT CALLS THE                        |  268 GET_NEW_CELL
127 |          GARBAGE COLLECTOR TO LOOK FOR SOME                "                                   |  269 GET_NEW_CELL
127 |                                                                                                 |  270 GET_NEW_CELL
127 |          IF LIST_POINTER=FSLP THEN                                                              |  271 GET_NEW_CELL
129 |          GARBAGE_COLLECT()                                                                      |  272 GET_NEW_CELL
130 |          ELSE                                                                                   |  273 GET_NEW_CELL
130 |          LIST_POINTER:=LIST_POINTER+1;                                                          |  274 GET_NEW_CELL
131 |       END = FREE_SPACE_LIST(.LIST_POINTER.);                                                    |  275 GET_NEW_CELL
131 |                                                                                                 |  276
131 |       FUNCTION GET_PARAMETER();                                                                 |  277
133 |                                                                                                 |  278 GET_PARAMETER
133 |       "  GETS THE NEXT ITEM FROM LIST POINTED TO BY CREG AND UPDATES CREG    "                 |  279 GET_PARAMETER
133 |                                                                                                 |  280 GET_PARAMETER
133 |       BITS(WORD_SIZE) TEMP;                                                                     |  281 GET_PARAMETER
COMPILER DIRECTIVE ...SCRATCH TEMP
134 |          TEMP:=LIST(.CREG.).HEAD;              CREG:=LIST(.CREG.).TAIL;                         |  282 GET_PARAMETER
137 |       END = TEMP;                                                                               |  283 GET_PARAMETER
137 |                                                                                                 |  284
137 |                                                                                                 |  285
137 |                                                                                                 |  286
COMPILER DIRECTIVE ...PAGE
```

```
137 |       PROCEDURE CHECK_TYPE(BITS(TAG_SIZE) TYPE;);                                  |  287
139 |                                                                                     |  288 CHECK_TYPE
139 |       "     CHECKS THAT THE TYPE OF TOPS_TAG IS THE SAME AS ITS PARAMETER          |  289 CHECK_TYPE
139 |          IF IT ISN'T THE INTERRUPT BIT INVALID TYPE IS SET       "                 |  290 CHECK_TYPE
139 |                                                                                     |  291 CHECK_TYPE
139 |          IF TOPS_TAG ¬= TYPE THEN                                                   |  292 CHECK_TYPE
141 |          $(  SET(ANY_INTERRUPT);  SET(INVALID_TYPE);  INTERRUPT_TYPE:=2;   $)      |  293 CHECK_TYPE
145 |       END;                                                                          |  294 CHECK_TYPE
145 |                                                                                     |  295
146 |                                                                                     |  296
146 | PROCEDURE CHECK_TYPES(BITS(TAG_SIZE) TYPE;);                                        |  297
143 |                                                                                     |  298 CHECK_TYPES
143 |       "   CHECKS THAT THE TYPES OF TOPS AND SECS CONFORM TO                        |  299 CHECK_TYPES
148 |           THE TYPE SPECIFIED   "                                                    |  300 CHECK_TYPES
148 |                                                                                     |  301 CHECK_TYPES
148 |       IF TOPS_TAG ¬= TYPE OR SECS_TAG ¬= TYPE THEN                                  |  302 CHECK_TYPES
150 |          $(                                                                         |  303 CHECK_TYPES
151 |               SET(ANY_INTERRUPT);   SET(INVALID_TYPE);                              |  304 CHECK_TYPES
153 |               INTERRUPT_TYPE:=2;                                                    |  305 CHECK_TYPES
154 |          $)                                                                         |  306 CHECK_TYPES
155 | END      " CHECK TYPES" ;                                                           |  307 CHECK_TYPES
155 |                                                                                     |  308
155 |                                                                                     |  309
COMPILER DIRECTIVE ...PAGE
```

```
155 |       "            THE INSTRUCTION PROCEDURES                    "                    |  310
155 |                                                                                         |  311
155 |       PROCEDURE COMMA();                                                               |  312
157 |                                                                                         |  313 COMMA
157 |       "   THE TOP STACK ELEMENT IS A LIST. GET A NEW LIST CELL AND INSERT               |  314 COMMA
157 |       TOPS_DATA AND SECS_DATA INTO THE TAIL AND HEAD RESPECTIVELY                      |  315 COMMA
157 |         THIS CORRESPONDS TO THE COMMA(,) OPERATOR IN SASL FOR MAKING LISTS     "       |  316 COMMA
157 |                                                                                         |  317 COMMA
157 |       BITS(WORD_SIZE) NEWCELL;                                                         |  318 COMMA
COMPILER DIRECTIVE ...SCRATCH NEWCELL
158 |          CHECK_TYPE(POINTER_TYPE);                                                     |  319 COMMA
160 |                NEWCELL:=GET_NEW_CELL();                                                 |  320 COMMA
161 |                LIST(.NEWCELL.).TAIL:=TOPS_DATA;                                         |  321 COMMA
162 |                LIST(.NEWCELL.).TAG:=SECS_TAG;                                           |  322 COMMA
163 |                LIST(.NEWCELL.).HEAD:=SECS_DATA;                                         |  323 COMMA
164 |                PUSHSTACK(POINTER_TYPE,NEWCELL);                                         |  324 COMMA
165 |       END;                                                                              |  325 COMMA
165 |                                                                                         |  326
165 |       PROCEDURE APPLY();                                                               |  327
167 |                                                                                         |  328 APPLY
167 |       "  THE SECOND STACK ELEMENT IS OF TYPE FUNCTION , THE TOP ELEMENT IS            |  329 APPLY
167 |          ITS ARGUEMENT.REMOVE THE TOP ELEMENT CHANGE CREG AND EREG TO                  |  330 APPLY
167 |          POINT TO THE FUNCTION, SAVING OLD VALUES ON THE STACK AND PUT                 |  331 APPLY
167 |          THE ARGUEMENT BACK ON THE STACK                 "                             |  332 APPLY
167 |                                                                                         |  333 APPLY
167 |          BITS(WORD_SIZE) LP;                                                            |  334 APPLY
169 |          IF SECS_TAG ¬= FUNCTION_TYPE THEN                                              |  335 APPLY
170 |             $(  SET(ANY_INTERRUPT);   SET(INVALID_TYPE);  EXIT;  $)                     |  336 APPLY
175 |          ELSE                                                                          |  337 APPLY
175 |             $(                                                                         |  338 APPLY
176 |                LP:=SECS_DATA;                                                           |  339 APPLY
177 |                SAVE_MACHINE_STATE                                                       |  340 APPLY
182 |                CREG:=LIST(.LP.).HEAD;                                                   |  341 APPLY
183 |                EREG:=LIST(.LP.).TAIL;                                                   |  342 APPLY
184 |             $)                                                                         |  343 APPLY
185 |       END;                                                                              |  344 APPLY
185 |                                                                                         |  345
185 |                                                                                         |  346
185 |                                                                                         |  347
185 |                                                                                         |  348
COMPILER DIRECTIVE ...PAGE
```

```
185 |       "  THE NEXT 3 PROCEDURES ARE CONCERNED WITH THE TEST EQUALS INSTRUCTION      |  349
185 |          WHICH CAN TEST ANYTHING , INCLUDING LISTS FOR EQUALITY. FUNCTION ARE      |  350
185 |          EXCEPTED.TRYING TO TEST FUNCTION EQUALITY CAUSES AN INTERRUPT             |  351
185 |          THE FUNCTION COMPARE TESTS THE TOP STACK VALUES FOR ABSOLUTE EQUALITY     |  352
185 |          BUT IF THEY ARE LISTS THE RECURSIVE PROCEDURE COMPARE_LISTS IS CALLED     |  353
185 |          WHICH CHAINS DOWN THE LISTS TESTING EACH ELEMENT IN TURN    "           |  354
185 |                                                                                    |  355
185 |                                                                                    |  356
185 |                                                                                    |  357
185 | PROCEDURE TEST_FOR_EQUALITY();                                                     |  358
187 |                                                                                    |  359 TEST_FOR_EQUALITY
187 |     "   TESTS THE TOP 2 STACK ELEMENTS FOR EQUALITY IF THEY ARE NOT              |  360 TEST_FOR_EQUALITY
187 |         LISTS.   SECS_DATA CONTAINS THE RESULT OF THE TEST   "                   |  361 TEST_FOR_EQUALITY
187 |                                                                                    |  362 TEST_FOR_EQUALITY
187 |     IF TOPS_TAG=FUNCTION_TYPE OR SECS_TAG=FUNCTION_TYPE THEN                      |  363 TEST_FOR_EQUALITY
189 |        $(                                                                          |  364 TEST_FOR_EQUALITY
191 |             SET(ANY_INTERRUPT);   INTERRUPT_TYPE:=4;                              |  365 TEST_FOR_EQUALITY
192 |        $)                                                                          |  366 TEST_FOR_EQUALITY
193 |     ELSE                                                                           |  367 TEST_FOR_EQUALITY
193 |           IF TOPS_TAG = SECS_TAG THEN                                              |  368 TEST_FOR_EQUALITY
194 |             $(                                                                     |  369 TEST_FOR_EQUALITY
195 |                  IF TOPS_DATA = SECS_DATA THEN                                     |  370 TEST_FOR_EQUALITY
196 |                     SECS_DATA:=LTRUE                                               |  371 TEST_FOR_EQUALITY
197 |                  ELSE                                                              |  372 TEST_FOR_EQUALITY
197 |                     SECS_DATA:=LFALSE;                                             |  373 TEST_FOR_EQUALITY
198 |             $)                                                                     |  374 TEST_FOR_EQUALITY
199 |       SECS_TAG:=LOGICAL_TYPE;                                                      |  375 TEST_FOR_EQUALITY
200 | END;  "  TEST_FOR_EQUALITY  "                                                     |  376 TEST_FOR_EQUALITY
```

```
200 | PROCEDURE COMPARE_LISTS();                                                                    |  379
202 |                                                                                               |  380 COMPARE_LISTS
202 |     "  RECURSIVE PROCEDURE COMPARING ALL THE ELEMENTS OF A LIST  "                            |  381 COMPARE_LISTS
202 |                                                                                               |  382 COMPARE_LISTS
203 |     BITS(WORD_SIZE) PL1,PL2;                                                                 |  383 COMPARE_LISTS
203 |      REPEAT                                                                                   |  384 COMPARE_LISTS
205 |        $(                                                                                     |  385 COMPARE_LISTS
206 |     PL1:=TOPS_DATA;           PL2:=SECS_DATA;                                                 |  386 COMPARE_LISTS
208 |     TOPS_TAG:=LIST(.PL1.).TAG;     TOPS_DATA:=LIST(.PL1.).HEAD;                               |  387 COMPARE_LISTS
210 |     SECS_TAG:=LIST(.PL2.).TAG;      SECS_DATA:=LIST(.PL2.).HEAD;                              |  388 COMPARE_LISTS
212 |     PL1:=LIST(.PL1.).TAIL;          PL2:=LIST(.PL2.).TAIL;                                    |  389 COMPARE_LISTS
214 |     IF TOPS_TAG = POINTER_TYPE AND SECS_TAG = POINTER_TYPE THEN                               |  390 COMPARE_LISTS
215 |        $(                                                                                     |  391 COMPARE_LISTS
216 |             SAVE_VALUES_OF_POINTERS                                                           |  392 COMPARE_LISTS
217 |             COMPARE_LISTS();                                                                  |  393 COMPARE_LISTS
220 |             RESTORE_VALUES_OF_POINTERS                                                        |  394 COMPARE_LISTS
223 |        $)                                                                                     |  395 COMPARE_LISTS
224 |     ELSE                                                                                      |  396 COMPARE_LISTS
224 |        TEST_FOR_EQUALITY();                                                                   |  397 COMPARE_LISTS
225 |     $)                                                                                        |  398 COMPARE_LISTS
224 |      UNTIL                                                                                    |  399 COMPARE_LISTS
226 |             PL1=NIL AND PL2=NIL AND SECS_TAG=LOGICAL_TYPE OR SECS_DATA=                       |  400 COMPARE_LISTS
226 |             LFALSE DO NULL;                                                                   |  401 COMPARE_LISTS
227 | END;    " COMPARE_LISTS   "                                                                   |  402 COMPARE_LISTS
227 |                                                                                               |  403
227 |                                                                                               |  404
227 |                                                                                               |  405
COMPILER DIRECTIVE ---PAGE--
```

```
227 | PROCEDURE TEST_EQUALS();                                                                  |  406
228 |                                                                                           |  407 TEST_EQUALS
229 |     "  TESTS THE TOP 2 STACK ELEMENTS FOR EQUALITY   "                                    |  408 TEST_EQUALS
230 |                                                                                           |  409 TEST_EQUALS
229 |     IF TOPS_TAG = POINTER_TYPE AND SECS_TAG=POINTER_TYPE THEN                             |  410 TEST_EQUALS
231 |        COMPARE_LISTS()                                                                    |  411 TEST_EQUALS
232 |     ELSE                                                                                  |  412 TEST_EQUALS
232 |        TEST_FOR_EQUALITY();                                                               |  413 TEST_EQUALS
233 |     UNSET(TOPSF);                                                                         |  414 TEST_EQUALS
234 | END;   "  TEST_EQUALS  "                                                                  |  415 TEST_EQUALS
234 |                                                                                           |  416
234 |                                                                                           |  417
234 |                                                                                           |  418
234 |      "    THE NEXT LOT OF PROCEDURES TAKE ONE OPERAND FROM THE STACK                      |  419
234 |        AND PUT ONE BACK ON            "                                                   |  420
234 |                                                                                           |  421
234 |      PROCEDURE HEAD();                                                                    |  422
235 |                                                                                           |  423 HEAD
235 |      "  TOP STACK IS A LIST PUT THE HEAD OF THIS LIST ON THE STACK   "                     |  424 HEAD
235 |                                                                                           |  425 HEAD
236 |         CHECK_TYPE(POINTER_TYPE);                                                         |  426 HEAD
237 |        TOPS_TAG:=LIST(.TOPS_DATA.).TAG;                                                   |  427 HEAD
239 |        TOPS_DATA:=LIST(.TOPS_DATA.).HEAD;                                                 |  428 HEAD
240 |      END;                                                                                 |  429 HEAD
240 |                                                                                           |  430
240 |      PROCEDURE TAIL();                                                                    |  431
242 |                                                                                           |  432 TAIL
242 |      "     SAME AS HEAD BUT PUTS TAIL ON STACK       "                                    |  433 TAIL
242 |                                                                                           |  434 TAIL
242 |         CHECK_TYPE(POINTER_TYPE);                                                         |  435 TAIL
244 |         TOPS_TAG:=POINTER_TYPE;                                                           |  436 TAIL
245 |         TOPS_DATA:=LIST(.TOPS_DATA.).TAIL;                                                |  437 TAIL
246 |      END;                                                                                 |  438 TAIL
246 |                                                                                           |  439
246 |      PROCEDURE TEST_STACK_TYPE(BITS(TAG_SIZE) TYPE;);                                     |  440
248 |                                                                                           |  441 TEST_STACK_TYPE
248 |      " IF THE TOP STACK TYPE IS THE SAME AS PARAMETER PUSHES TRUE ELSE FALSE    "         |  442 TEST_STACK_TYPE
248 |                                                                                           |  443 TEST_STACK_TYPE
248 |         IF TOPS_TAG ¬= TYPE THEN                                                          |  444 TEST_STACK_TYPE
250 |            TOPS_DATA:=LFALSE                                                              |  445 TEST_STACK_TYPE
```

```
251 |          ELSE                                                                  |  446 TEST_STACK_TYPE
251 |            TOPS_DATA:=LTRUE;                                                   |  447 TEST_STACK_TYPE
252 |      TOPS_TAG:=LOGICAL_TYPE;                                                   |  448 TEST_STACK_TYPE
253 |       END;                                                                      |  449 TEST_STACK_TYPE
253 |                                                                                 |  450
253 |                                                                                 |  451
253 | PROCEDURE DIGIT_OR_LETTER(BITS(1) DIGIT_TEST;);                                 |  452
255 |                                                                                 |  453 DIGIT_OR_LETTER
255 |    "  TESTS IF THE TOPSTACK ELEMENT IS EITHER A DIGIT OR                       |  454 DIGIT_OR_LETTER
255 |       A LETTER   "                                                              |  455 DIGIT_OR_LETTER
255 |                                                                                 |  456 DIGIT_OR_LETTER
255 |    FLAG TFLAG;                                                                 |  457 DIGIT_OR_LETTER
256 |    CHECK_TYPE(CHAR_TYPE);     " IT MUST BE A CHARACTER  "                       |  458 DIGIT_OR_LETTER
258 |    IF DIGIT_TEST=1 AND TOPS_DATA > 239 THEN                                     |  459 DIGIT_OR_LETTER
259 |       SET(TFLAG)                                                               |  460 DIGIT_OR_LETTER
260 |    ELSE                                                                         |  461 DIGIT_OR_LETTER
260 |       IF DIGIT_TEST=0 AND TOPS_DATA > 192 AND TOPS_DATA < 234 THEN              |  462 DIGIT_OR_LETTER
261 |       SET(TFLAG);                                                              |  463 DIGIT_OR_LETTER
262 |    IF TRUE(TFLAG) THEN                                                          |  464 DIGIT_OR_LETTER
263 |       TOPS_DATA:=LTRUE;                                                        |  465 DIGIT_OR_LETTER
264 |     ELSE                                                                        |  466 DIGIT_OR_LETTER
264 |       TOPS_DATA:=LFALSE;                                                       |  467 DIGIT_OR_LETTER
265 |    TOPS_TAG:=LOGICAL_TYPE;                                                     |  468 DIGIT_OR_LETTER
265 | END  " DIGIT OR LETTER  ";                                                      |  469 DIGIT_OR_LETTER
265 |                                                                                 |  470
265 |                                                                                 |  471
COMPILER DIRECTIVE ...PAGE
```

```
266 |       PROCEDURE DIGITVAL();                                                                  |  472
269 |                                                                                              |  473 DIGITVAL
268 |       "  CONVERTS THE TOP STACK TO INTEGER TYPE FROM CHAR TYPE IF IT IS                     |  474 DIGITVAL
269 |             A DIGIT                "                                                      |  475 DIGITVAL
268 |                                                                                              |  476 DIGITVAL
268 |          CHECK_TYPE(CHAR_TYPE);                                                              |  477 DIGITVAL
270 |         IF TOPS_DATA > 239 THEN                                                              |  478 DIGITVAL
271 |        $(                                                                                    |  479 DIGITVAL
272 |            TOPS_TAG:=INTEGER_TYPE;                                                           |  480 DIGITVAL
273 |            TOPS_DATA:=TOPS_DATA - 240;                                                       |  481 DIGITVAL
274 |        $)                                                                                    |  482 DIGITVAL
275 |       END;                                                                                   |  483 DIGITVAL
275 |                                                                                              |  484
275 |       PROCEDURE NOTTOP();                                                                    |  485
277 |                                                                                              |  486 NOTTOP
277 |       "  COMPLEMENTS THE TOP STACK ELEMENT       "                                         |  487 NOTTOP
277 |                                                                                              |  488 NOTTOP
277 |          CHECK_TYPE(LOGICAL_TYPE);                                                           |  489 NOTTOP
279 |          TOPS_DATA:= ¬TOPS_DATA;                                                             |  490 NOTTOP
280 |       END;                                                                                   |  491 NOTTOP
280 |                                                                                              |  492
280 |       PROCEDURE NEGPOS();                                                                    |  493
282 |                                                                                              |  494 NEGPOS
282 |       "  PRODUCES THE TWO'S COMPLEMENT OF THE TOP STACK ELEMENT    "                       |  495 NEGPOS
282 |                                                                                              |  496 NEGPOS
282 |          CHECK_TYPE(INTEGER_TYPE);                                                           |  497 NEGPOS
284 |         TOPS_DATA:= ¬TOPS_DATA + 1;                                                          |  498 NEGPOS
285 |       END;                                                                                   |  499 NEGPOS
285 |                                                                                              |  500
285 |       PROCEDURE FORK();                                                                      |  501
287 |                                                                                              |  502 FORK
287 |        "     EXECUTES A CODE BRANCH      "                                                   |  503 FORK
287 |          CHECK_TYPE(LOGICAL_TYPE);                                                           |  504 FORK
289 |                IF TOPS_DATA = LTRUE THEN   CREG:=LIST(.CREG.).HEAD                           |  505 FORK
291 |                ELSE  CREG:=LIST(.CREG.).TAIL;                                                |  506 FORK
292 |       END;                                                                                   |  507 FORK
293 |                                                                                              |  508
```

```
292 |       "     THE NEXT LOT OF INSTRUCTION PROCEDURES ARE FOR THOSE INSTRUCTIONS    |  509
292 |          WHICH HAVE A PARAMETER          "                                      |  510
292 |                                                                                 |  511
292 |       PROCEDURE LOADFN();                                                       |  512
294 |                                                                                 |  513 LOADFN
294 |       "  PARAMETER IS A POINTER TO CODE. CREATE A FUNCTION WITH THIS AS        |  514 LOADFN
294 |             THE FUNCTION CODE AND EREG POINTING TO IT'S ENVIRONMENT            |  515 LOADFN
296 |             PUSH A POINTER TO THIS FUNCTION ONTO THE STACK    "                |  516 LOADFN
294 |                                                                                 |  517 LOADFN
294 |          BITS(WORD_SIZE) NEWCELL;                                               |  518 LOADFN
COMPILER DIRECTIVE ...SCRATCH NEWCELL
295 |          NEWCELL:=GET_NEW_CELL();                                               |  519 LOADFN
297 |          LIST(.NEWCELL.).TAG:=FUNCTION_TYPE;                                    |  520 LOADFN
298 |          LIST(.NEWCELL.).HEAD:=GET_PARAMETER();                                 |  521 LOADFN
299 |          LIST(.NEWCELL.).TAIL:=EREG;                                            |  522 LOADFN
300 |          PUSHSTACK(FUNCTION_TYPE,NEWCELL);                                      |  523 LOADFN
301 |       END;                                                                      |  524 LOADFN
301 |                                                                                 |  525
301 |       PROCEDURE BLOCK();                                                        |  526
303 |                                                                                 |  527 BLOCK
303 |       "  AN UNCONDITIONAL BRANCH , BUT SAVING THE MACHINE STATE  "             |  528 BLOCK
303 |          "       PARAMETER GIVES NEW VALUE OF CREG          "                   |  529 BLOCK
303 |                                                                                 |  530 BLOCK
303 |          SAVE_MACHINE_STATE                                                     |  531 BLOCK
307 |          CREG:=LIST(.CREG.).HEAD;                                               |  532 BLOCK
310 |       END;                                                                      |  533 BLOCK
310 |                                                                                 |  534
310 |       PROCEDURE RETURN();                                                       |  535
312 |       "     IF STACK HAS ONE ELEMENT WE ARE FINISHED OTHERWISE                  |  536 RETURN
312 |          TOP STACK IS THE RESULT OF A FUNCTION , RESTORE MACHINE STATE         |  537 RETURN
312 |             THEN PUSH RESULT BACK ONTO THE STACK          "                    |  538 RETURN
312 |                                                                                 |  539 RETURN
312 |          BITS(TAG_SIZE) T1;          BITS(WORD_SIZE) T2;                        |  540 RETURN
314 |          IF STACK_POINTER=0 AND FALSE(TOPSF) THEN                               |  541 RETURN
316 |             SET(ENDPROG)                                                        |  542 RETURN
317 |          ELSE                                                                   |  543 RETURN
317 |           RESTORE_MACHINE_STATE                                                 |  544 RETURN
323 |       END;                                                                      |  545 RETURN
323 |                                                                                 |  546
```

```
389 |                       " END OF HIGHEST LEVEL LIST -- EXIT  "                    |  654 FLATTEN
389 |                                                                                 |  655 FLATTEN
389 |                        EXIT;                                                    |  656 FLATTEN
390 |                $)                                                               |  657 FLATTEN
391 |             ELSE                                                                |  658 FLATTEN
391 |                $(                                                               |  659 FLATTEN
392 |                                                                                 |  660 FLATTEN
392 |                       "  ELEMENT IS.NOT A LIST SO CHAIN TO NEXT ELEMENT  "     |  661 FLATTEN
392 |                                                                                 |  662 FLATTEN
392 |                       OP:=P;                                                    |  663 FLATTEN
393 |                       P:=LIST(.P.).TAIL;                                        |  664 FLATTEN
394 |                $)                                                               |  665 FLATTEN
395 |                                                                                 |  666 FLATTEN
395 |          "   CALL FLATTEN RECURSIVELY ALTHOUGH IF NOT A LIST THIS IS JUST      |  667 FLATTEN
395 |              A MEANS OF REPEATING THE PROCEDURE    "                           |  668 FLATTEN
395 |                                                                                 |  669 FLATTEN
395 |          FLATTEN(P,OP);                                                         |  670 FLATTEN
396 |                                                                                 |  671 FLATTEN
396 |  END   " FLATTEN ";                                                             |  672 FLATTEN
395 |                                                                                 |  673
395 |                                                                                 |  674
COMPILER DIRECTIVE ...PAGE
```

```
323 |       PROCEDURE HEADCONST();                                                                  |  547
325 |                                                                                               |  548 HEADCONST
325 |       "  PARAMETER IS A CONSTANT -- PUSH IT ONTO THE STACK     "                            |  549 HEADCONST
325 |                                                                                               |  550 HEADCONST
325 |          PUSHSTACK(LIST(.CREG.).TAG,LIST(.CREG.).HEAD);                                      |  551 HEADCONST
327 |          CREG:=LIST(.CREG.).TAIL;                                                             |  552 HEADCONST
328 |       END;                                                                                    |  553 HEADCONST
328 |                                                                                               |  554
328 |       PROCEDURE LOOKUP();                                                                     |  555
330 |                                                                                               |  556 LOOKUP
330 |       "  THIS PROCEDURE SEARCHES THE ENVIRONMENT FOR A NAME THE SAME AS ITS                  |  557 LOOKUP
330 |             PARAMETER. WHEN FOUND PUSH VALUE ASSOCIATED WITH THIS NAME ONTO THE              |  558 LOOKUP
330 |                STACK                    "                                                     |  559 LOOKUP
330 |                                                                                               |  560 LOOKUP
330 |          BITS(WORD_SIZE) T1,T2,T3,T4;                                                         |  561 LOOKUP
COMPILER DIRECTIVE ...SCRATCH T1,T2,T3,T4
331 |          T1:=EREG;              T2:=GET_PARAMETER();                                          |  562 LOOKUP
334 |       REPEAT                                                                                 |  563 LOOKUP
335 |             $(                                                                               |  564 LOOKUP
336 |                T3:=LIST(.T1.).HEAD;        T4:=LIST(.T1.).TAIL;                               |  565 LOOKUP
338 |                T1:=LIST(.T4.).TAIL;                                                           |  566 LOOKUP
339 |             $)                                                                               |  567 LOOKUP
340 |          UNTIL T2 = T3 DO                                                                    |  568 LOOKUP
340 |                NULL;                                                                         |  569 LOOKUP
341 |          PUSHSTACK(LIST(.T4.).TAG,LIST(.T4.).HEAD);                                          |  570 LOOKUP
342 |       END;                                                                                    |  571 LOOKUP
342 |                                                                                               |  572
```

```
342 |       PROCEDURE ADD_NAME_AND_VALUE_TO_ENVIRONMENT();                                  |  573
344 |                                                                                        |  574 ADD_NAME_AND_VALUE_T
344 |       "   THIS PROCEDURE ADDS A NAME AND A VALUE TO THE FRONT OF THE                  |  575 ADD_NAME_AND_VALUE_T
344 |          ENVIRONMENT LIST . SECS_DATA POINTS TO THE NAME AND TOPS_DATA                 |  576 ADD_NAME_AND_VALUE_T
344 |             POINTS TO THE VALUE                "                                       |  577 ADD_NAME_AND_VALUE_T
344 |                                                                                        |  578 ADD_NAME_AND_VALUE_T
344 |          BITS(WORD_SIZE) NC1,NC2;                                                      |  579 ADD_NAME_AND_VALUE_T
COMPILER DIRECTIVE ...SCRATCH NC1,NC2
345 |          NC1:=GET_NEW_CELL();                NC2:=GET_NEW_CELL();                      |  580 ADD_NAME_AND_VALUE_T
349 |          LIST(.NC1.).TAG:=LIST(.SECS_DATA.).TAG;                                       |  581 ADD_NAME_AND_VALUE_T
349 |          LIST(.NC1.).HEAD:=LIST(.SECS_DATA.).HEAD;                                     |  582 ADD_NAME_AND_VALUE_T
350 |          LIST(.NC1.).TAIL:=NC2;                                                        |  583 ADD_NAME_AND_VALUE_T
351 |                                                                                        |  584 ADD_NAME_AND_VALUE_T
351 |          "   IF NOT A GUESS TYPE ENTER VALUE ELSE SET TAG TO GUESS TYPE      "       |  585 ADD_NAME_AND_VALUE_T
351 |                                                                                        |  586 ADD_NAME_AND_VALUE_T
351 |          IF FALSE(GUESS_FLAG) THEN                                                    |  587 ADD_NAME_AND_VALUE_T
352 |             $(                                                                         |  588 ADD_NAME_AND_VALUE_T
353 |                LIST(.NC2.).TAG:=LIST(.TOPS_DATA.).TAG;                                 |  589 ADD_NAME_AND_VALUE_T
354 |                LIST(.NC2.).HEAD:=LIST(.TOPS_DATA.).HEAD;                               |  590 ADD_NAME_AND_VALUE_T
355 |             $)                                                                         |  591 ADD_NAME_AND_VALUE_T
356 |          ELSE                                                                          |  592 ADD_NAME_AND_VALUE_T
356 |             LIST(.NC2.).TAG:=GUESS_TYPE;                                               |  593 ADD_NAME_AND_VALUE_T
357 |          LIST(.NC2.).TAIL:=EREG;                                                       |  594 ADD_NAME_AND_VALUE_T
358 |          EREG:=NC1;                                                                    |  595 ADD_NAME_AND_VALUE_T
359 |       END;                                                                             |  596 ADD_NAME_AND_VALUE_T
359 |                                                                                        |  597
359 |                                                                                        |  598
```

```
359 |                                                                                                                    |  599
359 |       PROCEDURE CHANGE_GUESS_TO_ACTUAL_VALUE();                                                                   |  600
361 |                                                                                                                    |  601 CHANGE_GUESS_TO_ACTU
361 |       "     SEARCHES ENVIRONMENT TILL IT FINDS A NAME MATCHED BY THE                                               |  602 CHANGE_GUESS_TO_ACTU
361 |          NAME POINTED TO BY SECS_DATA . WHEN FOUND ASSOCIATES THE VALUE                                         |  603 CHANGE_GUESS_TO_ACTU
361 |             POINTED AT BY TOPS_DATA WITH THE NAME IN THE ENVIRONMENT     "                                     |  604 CHANGE_GUESS_TO_ACTU
361 |                                                                                                                    |  605 CHANGE_GUESS_TO_ACTU
361 |          BITS(WORD_SIZE) T1,T2;                                                                                    |  606 CHANGE_GUESS_TO_ACTU
COMPILER DIRECTIVE ...SCRATCH T1,T2
362 |          T1:=EREG;       T2:=LIST(.SECS_DATA.).HEAD;                                                           |  607 CHANGE_GUESS_TO_ACTU
365 |          WHILE LIST(.T1.).HEAD   = T2 DO                                                                       |  608 CHANGE_GUESS_TO_ACTU
366 |              T1:=LIST(.LIST(.T1.).TAIL.).TAIL;                                                                 |  609 CHANGE_GUESS_TO_ACTU
367 |          LIST(.LIST(.T1.).TAIL.).HEAD:=LIST(.TOPS_DATA.).HEAD;                                                 |  610 CHANGE_GUESS_TO_ACTU
368 |       END;                                                                                                         |  611 CHANGE_GUESS_TO_ACTU
368 |                                                                                                                    |  612


368 |                                                                                                                    |  613
COMPILER DIRECTIVE ...PAGE
```

```
PROCEDURE FLATTEN(BITS(WORD_SIZE) P,OP;);

    "   P POINTS TO A LIST AND OP POINTS TO THE PROVIOUS ELEMENT
        IN THE LIST IF P IS ITSELF THE MEMBER OF A LIST.  THE
        EFFECT OF FLATTEN IS TO FLATTEN THE LIST POINTED TO BY
        P  IE ENSURE THAT NONE OF THE ELEMENTS OF THIS LIST
        ARE THEMSELVES LISTS.  IF AN ELEMENT IS A LIST IT IS
        INCORPORATED IMNTO THE LIST POINTED TO BY P    "


         IF LIST(.P.).TAG=POINTER_TYPE THEN
           $(

               "    WE HAVE A SUBSIDIARY LIST.  CHAIN IT ONTO THE
                    PREVIOUS LIST WHOSE LAST ELEMENT IS POINTED TO
                    BY OP       "

                PUSHSTACK(POINTER_TYPE,P);
                P:=LIST(.P.).HEAD;
                LIST(.OP.).TAIL:=P;
                RECLEVEL:=RECLEVEL+1;
           $)
         ELSE
            IF LIST(.P.).TAIL=NIL THEN
              $(

                  " IF PROCESSING SUBSIDIARY LIST  CHAIN ITS LAST
                    ELEMENT ONTO THE NEXT ELEMENT IN THE HIGHER LEVEL
                    LIST    RESET P AND REDUCE RECLEVEL   "

                  IF RECLEVEL > 0 THEN
                    $(
                         RECLEVEL:=RECLEVEL-1;
                         FILL_TOPSTACK_REGISTERS();
                         OP:=P;       UNSET(TOPSF);
                         P:=LIST(.TOPS_DATA.).TAIL;
                         LIST(.OP.).TAIL:=P;
                      $)
                  ELSE
```

```
396 |       PROCEDURE DECL();                                                                  |  675
398 |                                                                                          |  676 DECL
398 |       "      PROCESS SASL DECLARATIONS   IF WE DON'T KNOW WHAT THEIR VALUES              |  677 DECL
398 |              ARE FILLS IN A GUESS VALUE TO BE MODIFIED LATER                             |  678 DECL
398 |              PARAMETER POINTS AT NAMES , TOP STACK POINTS AT VALUES   "                  |  679 DECL
398 |                                                                                          |  680 DECL
398 |               BITS(WORD_SIZE) OP;                                                        |  681 DECL
399 |               PUSHSTACK(LOGICAL_TYPE,GET_PARAMETER());                                  |  682 DECL
401 |               FLATTEN(TOPS_DATA,OP);                                                     |  683 DECL
402 |               IF FALSE(GUESS_FLAG) THEN                                                  |  684 DECL
403 |                  FLATTEN(SECS_DATA,OP);                                                  |  685 DECL
404 |               REPEAT                                                                     |  686 DECL
405 |                 $(                                                                       |  687 DECL
406 |                     ADD_NAME_AND_VALUE_TO_ENVIRONMENT();                                 |  688 DECL
407 |                                                                                          |  689 DECL
407 |                     TOPS_DATA:=LIST(.TOPS_DATA.).TAIL;                                   |  690 DECL
408 |                     IF FALSE(GUESS_FLAG) THEN                                            |  691 DECL
409 |                       SECS_DATA:=LIST(.SECS_DATA.).TAIL;                                 |  692 DECL
410 |                 $)                                                                       |  693 DECL
411 |               UNTIL                                                                      |  694 DECL
411 |                     TOPS_DATA = NIL   DO    NULL;                                        |  695 DECL
412 |       END" DECL ";                                                                      |  696 DECL
412 |                                                                                          |  697
413 |       PROCEDURE KNOT();                                                                 |  698
414 |                                                                                          |  699 KNOT
414 |       "      FILLS IN THE GUESS VALUES IN THE ENVIRONMENT                                |  700 KNOT
414 |          PARAMETER POINTS TO NAMES , TOP STACK TO VALUES        "                       |  701 KNOT
414 |                                                                                          |  702 KNOT
414 |          SECS_DATA:=GET_PARAMETER();                                                     |  703 KNOT
415 |          CHANGE_GUESS_TO_ACTUAL_VALUE();                                                |  704 KNOT
417 |       END " KNOT " ;                                                                    |  705 KNOT
417 |                                                                                          |  706
417 |                                                                                          |  707
417 | PROCEDURE PRINTELEMENT(BITS(TAG_SIZE) TAG; BITS(WORD_SIZE) DATA;);                       |  708
419 |                                                                                          |  709 PRINTELEMENT
419 |     IF TAG = INTEGER_TYPE THEN                                                           |  710 PRINTELEMENT
421 |        WRITENUMBER(DATA)                                                                 |  711 PRINTELEMENT
422 |     ELSE                                                                                 |  712 PRINTELEMENT
422 |        IF TAG = CHAR_TYPE THEN                                                           |  713 PRINTELEMENT
423 |          OUTPUT_BUFFER(.OUTBUFF_PNT.):=DATA                                              |  714 PRINTELEMENT
```

```
424 |         ELSE
424 |           IF TAG = LOGICAL_TYPE THEN
425 |               IF DATA = 0 THEN
426 |                  WRITESTRING('#FALSE#')
427 |               ELSE
427 |                  WRITESTRING('#TRUE#')
428 |           ELSE
428 |                WRITESTRING('#FUNCTION#');
429 | END  " PRINTELEMENT ";
429 |
429 |
429 |
429 | PROCEDURE PRINTRESULT();
431 |
431 |     BITS(WORD_SIZE) T;
432 |     IF TOPS_TAG = POINTER_TYPE THEN
434 |        $(
435 |            FLATTEN(T,TOPS_DATA);
436 |            REPEAT
437 |              $(
438 |               PRINTELEMENT(LIST(.TOPS_DATA.).TAG,LIST(.
439 |               TOPS_DATA:=LIST(.TOPS_DATA.).TAIL;
440 |              $)
441 |            UNTIL
441 |                TOPS_DATA = NIL  DO  NULL;
442 |        $)
443 |     ELSE
443 |        PRINTELEMENT(TOPS_TAG,TOPS_DATA);
444 |     WRITE();
445 | END " PRINTRESULT ";
445 |
445 |
445 |
COMPILER DIRECTIVE ...PAGE
```

```
                                        715 PRINTELEMENT
                                        716 PRINTELEMENT
                                        717 PRINTELEMENT
                                        718 PRINTELEMENT
                                        719 PRINTELEMENT
                                        720 PRINTELEMENT
                                        721 PRINTELEMENT
                                        722 PRINTELEMENT
                                        723 PRINTELEMENT
                                        724
                                        725
                                        726
                                        727
                                        728 PRINTRESULT
                                        729 PRINTRESULT
                                        730 PRINTRESULT
                                        731 PRINTRESULT
                                        732 PRINTRESULT
                                        733 PRINTRESULT
                                        734 PRINTRESULT
.TOPS_DATA.).TAIL);                     735 PRINTRESULT
                                        736 PRINTRESULT
                                        737 PRINTRESULT
                                        738 PRINTRESULT
                                        739 PRINTRESULT
                                        740 PRINTRESULT
                                        741 PRINTRESULT
                                        742 PRINTRESULT
                                        743 PRINTRESULT
                                        744 PRINTRESULT
                                        745
                                        746
                                        747
```

```
445 |      PROCEDURE MAINLOOP();                                                                  |  748
447 |                                                                                             |  749 MAINLOOP
447 |    "*************************************************************************             |  750 MAINLOOP
447 |                                                                                             |  751 MAINLOOP
447 |      THIS IS THE DRIVER PROCEDURE FOR THE SASL S MACHINE                                    |  752 MAINLOOP
447 |    S INSTRUCTIONS ARE FETCHED TILL ENDPROG IS SET, THE STACK IS POPPED                      |  753 MAINLOOP
447 |    FOR INSTRUCTIONS WHICH NEED IT ,AND IF INTERRUPTS HAVE OCCURRED WE                       |  754 MAINLOOP
447 |    HANDLE THEM                                                                              |  755 MAINLOOP
447 |                                                                                             |  756 MAINLOOP
447 |    *************************************************************************"            |  757 MAINLOOP
447 |                                                                                             |  758 MAINLOOP
447 |        FLAG TFLAG;     UNSET(TFLAG);                                                        |  759 MAINLOOP
450 |                                                                                             |  760 MAINLOOP
450 |        WHILE FALSE(ENDPROG) DO                                                              |  761 MAINLOOP
451 |           $(                                                                                |  762 MAINLOOP
452 |              IF TRUE(ANY_INTERRUPT) THEN       HANDLE_INTERRUPTS();                         |  763 MAINLOOP
454 |              OP_CODE:=GET_PARAMETER();                                                      |  764 MAINLOOP
455 |              FILL_TOPSTACK_REGISTERS();                                                     |  765 MAINLOOP
456 |              IF OP_CODE < 12 THEN                                                           |  766 MAINLOOP
457 |                 $(                                                                          |  767 MAINLOOP
458 |                IF OP_CODE < 10 THEN                                                         |  768 MAINLOOP
459 |                    CHECK_TYPES(INTEGER_TYPE)                                                |  769 MAINLOOP
460 |                ELSE                                                                         |  770 MAINLOOP
460 |                    CHECK_TYPES(LOGICAL_TYPE);                                               |  771 MAINLOOP
461 |                    CASE OP_CODE OF                                                          |  772 MAINLOOP
462 |                       SECS_DATA:=SECS_DATA / TOPS_DATA;                                     |  773 MAINLOOP
463 |                       SECS_DATA:=SECS_DATA REM TOPS_DATA;                                   |  774 MAINLOOP
464 |                       SECS_DATA:=SECS_DATA * TOPS_DATA;                                     |  775 MAINLOOP
465 |                       SECS_DATA:=SECS_DATA - TOPS_DATA;                                     |  776 MAINLOOP
466 |                     IF SECS_DATA > TOPS_DATA THEN                                           |  777 MAINLOOP
467 |                        SET(TFLAG);                                                          |  778 MAINLOOP
468 |                    IF SECS_DATA >= TOPS_DATA THEN                                           |  779 MAINLOOP
469 |                        SET(TFLAG);                                                          |  780 MAINLOOP
470 |                    IF SECS_DATA < TOPS_DATA THEN                                            |  781 MAINLOOP
471 |                        SET(TFLAG);                                                          |  782 MAINLOOP
472 |                    IF SECS_DATA <= TOPS_DATA THEN                                           |  783 MAINLOOP
473 |                        SET(TFLAG);                                                          |  784 MAINLOOP
474 |                       SECS_DATA:=TOPS_DATA & SECS_DATA;                                     |  785 MAINLOOP
475 |                       SECS_DATA:=TOPS_DATA | SECS_DATA;                                     |  786 MAINLOOP
476 |                    ENDCASE;                                                                 |  787 MAINLOOP
```

```
476 |                     IF TRUE(TFLAG) THEN                                    | 788 MAINLOOP
477 |                         SECS_DATA:=LTRUE                                   | 789 MAINLOOP
478 |                     ELSE                                                   | 790 MAINLOOP
479 |                         SECS_DATA:=LFALSE;                                 | 791 MAINLOOP
479 |                     IF OP_CODE > 10 THEN                                   | 792 MAINLOOP
480 |                         SECS_TAG:=LOGICAL_TYPE;                            | 793 MAINLOOP
481 |                     UNSET(TOPSF);                                          | 794 MAINLOOP
482 |                 $)                                                         | 795 MAINLOOP
483 |                                                                            | 796 MAINLOOP
483 | " ******** LOGICAL OPERATIONS HERE ******** "                              | 797 MAINLOOP
483 |                                                                            | 798 MAINLOOP
483 |             ELSE                                                           | 799 MAINLOOP
483 |                 IF OP_CODE < 27 THEN                                       | 800 MAINLOOP
484 |                     $(                                                     | 801 MAINLOOP
485 |                                                                            | 802 MAINLOOP
485 |                         CASE OP_CODE - 11 OF                               | 803 MAINLOOP
486 |                             COMMA();                                       | 804 MAINLOOP
487 |                             APPLY();                                       | 805 MAINLOOP
488 |                             HEAD();                                        | 806 MAINLOOP
489 |                             TEST_STACK_TYPE(LOGICAL_TYPE);                 | 807 MAINLOOP
490 |                             TEST_STACK_TYPE(CHAR_TYPE);                    | 808 MAINLOOP
491 |                             TEST_STACK_TYPE(POINTER_TYPE);                 | 809 MAINLOOP
492 |                             TEST_STACK_TYPE(FUNCTION_TYPE);                | 810 MAINLOOP
493 |                             DIGIT_OR_LETTER(0);          "DIGIT"           | 811 MAINLOOP
494 |                             DIGIT_OR_LETTER(1);          "LETTER"          | 812 MAINLOOP
495 |                             DIGITVAL();                                    | 813 MAINLOOP
496 |                             NEGPOS();                                      | 814 MAINLOOP
497 |                             NEGPOS();                                      | 815 MAINLOOP
498 |                             NOTTOP();                                      | 816 MAINLOOP
499 |                             FORK();                                        | 817 MAINLOOP
500 |                         ENDCASE;                                           | 818 MAINLOOP
500 |                     $)                                                     | 819 MAINLOOP
501 |                 ELSE                                                       | 820 MAINLOOP
501 |                     CASE OP_CODE - 26 OF                                   | 821 MAINLOOP
502 |                             TEST_FOR_EQUALITY();                           | 822 MAINLOOP
503 |                             $(                                             | 823 MAINLOOP
504 |                                 TEST_FOR_EQUALITY();                       | 824 MAINLOOP
```

```
505 |                                  TOPS_DATA:= ¬TOPS_DATA;                    |  825 MAINLOOP
506 |                               $)                                            |  826 MAINLOOP
507 |                              DECL();        "NOT GUESS"                    |  827 MAINLOOP
508 |                              $(      SET(GUESS_FLAG);                      |  828 MAINLOOP
509 |                              DECL();  $)           " GUESS"                 |  829 MAINLOOP
510 |                              <NOT();                                       |  830 MAINLOOP
511 |                              BLOCK();                                      |  831 MAINLOOP
512 |                           ENDCASE;                                         |  832 MAINLOOP
513 |                 $)                                                         |  833 MAINLOOP
514 |           END;                                                             |  834 MAINLOOP
515 |               MAINLOOP();                                                  |  835
516 |               PRINTRESULT();                                               |  836
517 |           ENDPROGRAM                                                       |  837
```

****

COMPILATION COMPLETE -- NO OF ERRORS = 0

1357 MICROINSTRUCTIONS WERE GENERATED

```
B1700 SIMULATOR V.2(V.1 ST ANDREWS)                    UNIVERSITY O

*****       1531 MICROINSTRUCTIONS GENERATED *****

*****  S-PROGRAM LOAD FROM FILE  .S.PROG  *****

####  LOADER COMMENTS  ####

THE SASL PROGRAM TO SUM THE ELEMENTS OF A LIST( GIVEN ON PAGE 185 )
WAS TRANSLATED BY HAND TO MACHINE CODE SUITABLE FOR THE SASL S-MACHINE.
THIS CODE IS SHOWN BELOW

 1      BLOCK     15
 2
 3      DECLGUESS SUMLIST
 4      DECL LIST
 5      LDC NIL
 6      TL LIST
 7      EQ
 8      FORK      9,10
 9      HD LIST   " FOLLOWED BY NIL POINTER TO INDICATE END OF FUNCTION "
10      HD LIST
11      TL LIST
12      LOADFN SUMLIST
13      APPLY
14      PLUS    " FOLLOWED BY NIL POINTER TO INDICATE END OF FUNCTION "
15      LOADFN SUMLIST
16      TIEKNOT SUMLIST
17      LDC 1
18      LDC 2
19      COMMA
20      LDC 3
21      COMMA
22      LDC 4
23      COMMA
24      LDC 5
25      COMMA
26      LDC 6
27      COMMA
28      LDC 7
29      COMMA
30      LDC 8
31      COMMA
32      LDC 9
33      COMMA
34      LDC 10
```

```
35      COMMA
36      LOC NIL
37      COMMA
38      LOADFN SUMLIST
39      APPLY     " FOLLWED BY NIL POINTER "
```

***** 00000039 S-INSTRUCTIONS LOADED *****

B1700 SIMULATOR V.2(V.1 ST ANDREWS) UNIVERSITY O

***** EXECUTION *****

55

*** STATISTICAL SUMMARY HAS BEEN SUPPRESSED ***

NO OF CLOCK CYCLES 6357

***** END OF SIMULATION--- ALL MICRO INSTRUCTIONS PROCESSED **

****

****

0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 012345
7124-LPB+++....HERIOT-WATT.UNIVERSITY....B5700....U129025.U129025..U129025.U
7124-LPB+++....HERIOT-WATT.UNIVERSITY....B5700....U129025.U129025..U129025.U
7124-LPB+++....HERIOT-WATT.UNIVERSITY....B5700....U129025.U129025..U129025.U
7124-LPB+++....HERIOT-WATT.UNIVERSITY....B5700....U129025.U129025..U129025.U

```
SUILVEN COMPILER -- ST ANDREWS UNIVERSITY-- VERSION 24/8/75                    TODAY IS 03 MAY 77
SOURCE LANGUAGE : SNOBOL4(SPITBOL)          TARGET MACHINE : IBM 360/370
OPTIONS : ON = LIST. OFF = CODE,COPY,DUMP


1 |                                                                          |     1
1 |"*************************************************************          |     2
1 |                                                                          |     3
1 |   THIS IS THE SUILVEN CODE FOR THE PASCAL S MACHINE DESIGNED BY          |     4
1 |  WIRTH ET AL AT ZURICH. THIS MACHINE WAS WRITTEN INACCORDANCE WITH       |     5
1 |  THE PASCAL P COMPILER IMPLEMENTATION NOTES SUPPLIED BY WIRTH            |     6
1 |  PROGRAMMER : IAN SOMMERVILLE -- ST ANDREWS UNIVERSITY -- 23/8/75        |     7
1 |                                                                          |     8
1 |                                                                          |     9
1 |  THE OP CODES FOR THE P MACHINE INSTRUCTIONS FOLLOW                      |    10
1 |                                                                          |    11
1 |  OP CODE = 1       AND    BOOLEAN AND                                    |    12
1 |  OP CODE = 2       IOR    BOOLEAN INCLUSIVE OR                           |    13
1 |  OP CODE = 3       DIF    SET DIFFERENCE                                 |    14
1 |  OP CODE = 4       INT    SET INTERSECTION                               |    15
1 |  OP CODE = 5       INN    TEST SET MEMBERSHIP                            |    16
1 |  OP CODE = 6       UNI    SET UNION                                      |    17
1 |  OP CODE = 7       ADI    INTEGER ADDITION                               |    18
1 |  OP CODE = 8       SBI    INTEGER SUBTRACTION                            |    19
1 |  OP CODE = 9       MPI    INTEGER MULTIPLICATION                         |    20
1 |  OP CODE = 10      MOD    MODULUS                                        |    21
1 |  OP CODE = 11      DVI    INTEGER DIVISION                               |    22
1 |  OP CODE = 12      ADR    REAL ADDITION                                  |    23
1 |  OP CODE = 13      SBR    REAL SUBTRACTION                               |    24
1 |  OP CODE = 14      MPR    REAL MULTIPLICATION                            |    25
1 |  OP CODE = 15      DVR    REAL DIVISION                                  |    26
1 |  OP CODE = 16      EQU    TEST EQUALITY                                  |    27
1 |  OP CODE = 17      NEQ    TEST NOT EQUAL                                 |    28
1 |  OP CODE = 18      GEQ    TEST GREATER THAN OR EQUAL                     |    29
1 |  OP CODE = 19      GRT    TEST GREATER THAN                              |    30
1 |  OP CODE = 20      LEQ    TEST LESS THAN OR EQUALS                       |    31
1 |  OP CODE = 21      LES    TEST LESS THAN                                 |    32
1 |  OP CODE = 22      MOV    COPY ARRAYS                                    |    33
1 |  OP CODE = 23      IXA    COMPUTE INDEXED ADDRESS                        |    34
```

```
1 |     OP CODE = 24      NGI     INTEGER SIGN INVERSION                                  |  35
1 |     OP CODE = 25      NGR     REAL SIGN INVERSION                                     |  36
1 |     OP CODE = 26      NOT     BOOLEAN NOT                                             |  37
1 |     OP CODE = 27      FLT     FLOAT TOP OF STACK                                      |  38
1 |     OP CODE = 28      FLO     FLOAT NEXT TO TOP OF STACK                              |  39
1 |     OP CODE = 29      TRC     TRUNCATION                                              |  40
1 |     OP CODE = 30      ABI     ABSOLUTE VALUE OF INTEGER                               |  41
1 |     OP CODE = 31      ABR     ABSOLUTE VALUE OF REAL                                  |  42
1 |     OP CODE = 32      INC     INCREMENT ADDRESS                                       |  43
1 |     OP CODE = 33      SGS     GENERATE SINGLETON SET                                  |  44
1 |     OP CODE = 34      SQI     SQUARE INTEGER                                          |  45
1 |     OP CODE = 35      SQR     SQUARE REAL NUMBER                                      |  46
1 |     OP CODE = 36      ODD     TEST ON ODD                                             |  47
1 |     OP CODE = 37      DEC     DECREMENT ADDRESS                                       |  48
1 |     OP CODE = 38      LOD     LOAD CONTENTS OF ADDRESS                                |  49
1 |     OP CODE = 39      LDO     LOAD CONTENTS OF BASE LEVEL ADDRESS                     |  50
1 |     OP CODE = 40      LDA     LOAD ADDRESS                                            |  51
1 |     OP CODE = 41      LAO     LOAD BASE LEVEL ADDRESS                                 |  52
1 |     OP CODE = 42      LDC     LOAD CONSTANT                                           |  53
1 |     OP CODE = 43      LDCI    LOAD CONSTANT INDIRECT                                  |  54
1 |     OP CODE = 44      LCA     LOAD ADDRESS OF CONSTANT                                |  55
1 |     OP CODE = 45      STR     STORE AT ADDRESS                                        |  56
1 |     OP CODE = 46      SRO     STORE                                                   |  57
1 |     OP CODE = 47      STO     STORE AT BASE LEVEL ADDRESS                             |  58
1 |     OP CODE = 48      IND     INDEXED FETCH                                           |  59
1 |     OP CODE = 49      MST     MARK STACK                                              |  60
1 |     OP CODE = 50      CUP     CALL USER PROCEDURE                                     |  61
1 |     OP CODE = 51      ENT     ENTER BLOCK                                             |  62
1 |     OP CODE = 52      RET     RETURN FROM BLOCK                                       |  63
1 |     OP CODE = 53      CSP     CALL STANDARD PROCEDURE                                 |  64
1 |     OP CODE = 54      CHK     CHECK AGAINST UPPER AND LOWER BOUNDS                    |  65
1 |     OP CODE = 55      EOF     TEST ON END OF FILE                                     |  66
1 |     OP CODE = 56      UJP     UNCONDITIONAL JUMP                                      |  67
1 |     OP CODE = 57      XJP     INDEXED JUMP                                            |  68
1 |     OP CODE = 58      FJP     FALSE JUMP                                              |  69
1 |     OP CODE = 59      STP     STOP MACHINE                                            |  70
1 |                                                                                       |  71
1 | ********************************************************************************"    |  72
1 |                                                                                       |  73
1 |                                                                                       |  74
```

```
 1 |      MACRO 'WORD_SIZE' = '24';                                                    |  75
 2 |      MACRO 'P_SIZE' = '4';      MACRO 'Q_SIZE' = '14';                            |  76
 4 |      MACRO 'STACK_WIDTH'='24';                                                    |  77
 5 |      MACRO 'TOPSTORE'='6000';      MACRO 'OP_SIZE'='6';                           |  78
 7 |      MACRO 'LTRUE' = '1';         MACRO 'LFALSE' = '0';                           |  79
 9 |      MACRO 'MAXSTR' = '0';                                                        |  80
10 |      MACRO 'UNDEF' = '#FFFFFF';     MACRO 'LARGINT'='#EFFFFFE';                  |  81
12 |      MACRO 'INPUTADR' = '0';     MACRO 'PRDADR' = '0';                            |  82
14 |                                                                                   |  83
14 |      BITS(WORD_SIZE) ARRAY (TOPSTORE) STORE;                                      |  84
15 |      BITS(WORD_SIZE) ARRAY (64) INST_COUNT;                                       |  85
16 |      BITS(WORD_SIZE) STACKP,HEAPP,TOPS,SECS,ACC,ACC2,DATA_SEG_PNT,                |  86
17 |                      CINSTRREG,PROGC,INTERRUPT_TYPE;                              |  87
17 |      BITS(P_SIZE) P;     BITS(Q_SIZE) Q;     BITS(OP_SIZE) OP_CODE;               |  88
20 |                                                                                   |  89
20 |      FLAG UNINITIALISED_VALUE,STACK_OFLOW,VALUE_OUT_OF_RANGE,ENDPROG;            |  90
21 |      FLAG  VAL_TOO_BIG;                                                          |  91
22 |      TEMPLATE REALNO = CHAR(5),SIGN(1),MANTISSA(18);                             |  92
23 |      TEMPLATE INT = SIGN(1),NUMB(23);                                            |  93
24 |       FLAG ANY_INTERRUPT, TOPFULL,SECFULL;                                       |  94
OMPILER DIRECTIVE ...SCRATCH  STACKP,HEAPP,TOPS,SECS,ACC,ACC2,DATA_SEG_PNT,PROGC
OMPILER DIRECTIVE ...SCRATCH  CINSTRREG,INTERRUPT_TYPE
25 |                                                                                   |  95
OMPILER DIRECTIVE ...PAGE
```

```
25 |     PROCEDURE PUSH();                                                                  |  96
27 |                                                                                        |  97 PUSH
27 |     " THIS PROCEDURE IS USED TO PUSH A VALUE ONTO THE STACK_THE VALUE                  |  98 PUSH
27 |       TO BE PUSHED IS ALWAYS HELD IN THE REGISTER ACC. IT IS IMPLEMENTED               |  99 PUSH
27 |       THIS WAY RATHER THAN PASSED AS A PARAMETER FOR EFFICIENCY REASONS                | 100 PUSH
27 |       PUSH IS ONE OF THE MOST FREQUENTLY USED ROUTINES IN THE MACHINE                  | 101 PUSH
27 |       AND AS PARAMETER PASSING IS RELATIVELY INEFFICIENT IT IS AVOIDED    "           | 102 PUSH
27 |                                                                                        | 103 PUSH
27 |     IF TRUE(TOPFULL) THEN                                                              | 104 PUSH
20 |       $(                                                                               | 105 PUSH
30 |          STORE(.STACKP.):=SECS;                                                        | 106 PUSH
31 |          SECS:=TOPS;                                                                   | 107 PUSH
32 |          STACKP:=STACKP + 1;                                                           | 108 PUSH
33 |          TOPS:=ACC;                                                                    | 109 PUSH
34 |       $)                                                                               | 110 PUSH
35 |     ELSE                                                                               | 111 PUSH
35 |        IF TRUE(SECFULL) THEN                                                           | 112 PUSH
36 |           $(    TOPS:=ACC;      SET(TOPFULL);       $)                                | 113 PUSH
40 |        ELSE                                                                            | 114 PUSH
40 |           $(    SECS:=ACC;      SET(SECFULL);       $)                                | 115 PUSH
44 |     END;     "  PUSH "                                                                | 116 PUSH
44 |                                                                                        | 117
44 |                                                                                        | 118
44 |                                                                                        | 119
44 |     FUNCTION POP();                                                                    | 120
45 |                                                                                        | 121 POP
45 |     "  RETURNS THE TOP ELEMENT FROM THE STACK    "                                    | 122 POP
45 |                                                                                        | 123 POP
45 |     IF TRUE(TOPFULL) THEN                                                              | 124 POP
48 |        $(                                                                              | 125 POP
49 |          UNSET(TOPFULL);    EXIT = TOPS;                                               | 126 POP
51 |        $)                                                                              | 127 POP
52 |     ELSE                                                                               | 128 POP
52 |        IF TRUE(SECFULL) THEN                                                           | 129 POP
53 |          $(                                                                            | 130 POP
54 |             UNSET(SECFULL);     EXIT = SECS;                                           | 131 POP
56 |          $)                                                                            | 132 POP
57 |        ELSE                                                                            | 133 POP
57 |           STACKP:=STACKP - 1;                                                          | 134 POP
58 |     END = STORE(.STACKP.);                                                             | 135 POP
```

```
58 |                                                                                                  |  136
58 |                                                                                                  |  137
58 | PROCEDURE TEST(BITS(2) ACTION;);                                                                 |  138
60 |                                                                                                  |  139 TEST
60 |     FLAG TRUERESULT;  UNSET(TRUERESULT);                                                         |  140 TEST
63 |     CASE ACTION OF                                                                               |  141 TEST
64 |                                                                                                  |  142 TEST
64 |      "=" IF ACC = ACC2 THEN SET(TRUERESULT);                                                     |  143 TEST
65 |      ">=" IF ACC >= ACC2 THEN SET(TRUERESULT);                                                   |  144 TEST
68 |      "<=" IF ACC <= ACC2 THEN SET(TRUERESULT);                                                   |  145 TEST
70 |     ENDCASE;                                                                                     |  146 TEST
70 |     IF FALSE(TRUERESULT) THEN                                                                    |  147 TEST
71 |       ACC:=0                                                                                     |  148 TEST
72 |     ELSE                                                                                         |  149 TEST
72 |       ACC:=1;                                                                                    |  150 TEST
73 | END " TEST " ;                                                                                   |  151 TEST
73 |                                                                                                  |  152
73 | PROCEDURE HANDLE_INTERRUPT();                                                                    |  153
75 |                                                                                                  |  154 HANDLE_INTERRUPT
75 |     "  INTERRUPTS ARE NORMALLY HANDLED BY GISMO. THIS IS A DUMMY                                 |  155 HANDLE_INTERRUPT
75 |        ROUTINE WHICH MERELY STOPS PROCESSING "                                                   |  156 HANDLE_INTERRUPT
75 |                                                                                                  |  157 HANDLE_INTERRUPT
75 |      WRITESTRING(" INTERRUPT - JOB ZAPPED ");                                                    |  158 HANDLE_INTERRUPT
77 |      WRITE();     HALT;                                                                          |  159 HANDLE_INTERRUPT
79 | END " HANDLE_INTERRUPT ";                                                                        |  160 HANDLE_INTERRUPT
79 |                                                                                                  |  161
79 |                                                                                                  |  162
79 | FUNCTION NORMALISE(BITS(WORD_SIZE) T;);                                                          |  163
81 |                                                                                                  |  164 NORMALISE
81 |    " RETURNS THE PARAMETER T IN NORMALISED FORM  "                                               |  165 NORMALISE
81 |                                                                                                  |  166 NORMALISE
81 |    BITS(1) SB;   BITS(4) SHIFTS;   BITS(5)T1;                                                    |  167 NORMALISE
84 |    TEMPLATE RTOP = CHAR(5),SIGN(1),TOP(3),MANTT(15);                                             |  168 NORMALISE
85 |    DEFINE  RTOP : T;                                                                             |  169 NORMALISE
85 |    SB:=T.SIGN;  SHIFTS:=0;   T1:=T.CHAR;                                                         |  170 NORMALISE
90 |    WHILE T.TOP =0 DO                                                                             |  171 NORMALISE
91 |     $(                                                                                           |  172 NORMALISE
92 |         T := T SHL 3;   SHIFTS:=SHIFTS+1;                                                        |  173 NORMALISE
94 |         IF SB = 1 THEN                                                                           |  174 NORMALISE
95 |           T1:=T1+1                                                                               |  175 NORMALISE
```

```
            ELSE
               T1:=T1 - 1;
               IF SHIFTS = 6 THEN
                  EXIT = 0;
        $)
    T.SIGN:=SB;      T.CHAR:=T1;

END = T;


FUNCTION CVB(BITS(1)S;BITS(18) M;);

     " CONVERTS MANTISSA PART OF A REAL TO BINARY "

    BITS(24) OUTPUT,MULT;
    DEFINE INT : OUTPUT;
    OUTPUT := M & %111;     M:=M SHR 3;
    MULT:=8;
    REPEAT
       $(
           OUTPUT:=M & %111 * MULT + OUTPUT;
            M:=M SHR 3;     MULT:=MULT * 10;
        $)
    UNTIL
        M = 0 DO NULL;
    IF S = 1 THEN
       $(
           OUTPUT:= -OUTPUT + 1;
           OUTPUT.SIGN:=1;
       $)
END = OUTPUT;



FUNCTION CONVERT_TO_BINARY(BITS(WORD_SIZE) W;);

    "  CONVERTS OCTAL CHARACTERS TO BINARY EQUIVALENT  "

    DEFINE REALNO : W;
    BITS(24) OUTPUT,SHIFT;
```

```
176 NORMALISE
177 NORMALISE
178 NORMALISE
179 NORMALISE
180 NORMALISE
181 NORMALISE
182 NORMALISE
183 NORMALISE
184
185
186
187 CVB
188 CVB
189 CVB
190 CVB
191 CVB
192 CVB
193 CVB
194 CVB
195 CVB
196 CVB
197 CVB
198 CVB
199 CVB
200 CVB
201 CVB
202 CVB
203 CVB
204 CVB
205 CVB
206 CVB
207
208
209
210
211 CONVERT_TO_BINARY
212 CONVERT_TO_BINARY
213 CONVERT_TO_BINARY
214 CONVERT_TO_BINARY
215 CONVERT_TO_BINARY
```

```
    BITS(24) CHAR;
    DEFINE INT : OUTPUT;
    IF W.CHAR <= 16 THEN
      $(
       EXIT = 0;
      $)
    ELSE
       $(
            OUTPUT:=CVB(W.SIGN,W.MANTISSA);
            SHIFT := CHAR-16 * 3;
            IF OUTPUT.SIGN = 1 THEN
                OUTPUT.NUMB:=¬OUTPUT.NUMB+1;
            OUTPUT.NUMB:=OUTPUT.NUMB SHL SHIFT;
         $)
END = OUTPUT     " CONVERT_TO_BINARY " ;


FUNCTION CONV_TO_OCTAL(BITS(WORD_SIZE) A;);

"  CONVERTS BINARY TO OCTAL  "

     BITS(24) NUMB,DIG;  BITS(5) SHIFTS;
        DEFINE INT : A;
     NUMB:=0;    SHIFTS:=0;
     IF A.SIGN=1 THEN
      $(
          A.NUMB:=¬A.NUMB+1;
          A.SIGN:=0;
     $)
     REPEAT
       $(
            DIG:=A REM 8;    A:=A/8;
            NUMB:=DIG SHL SHIFTS | NUMB;
            SHIFTS:=SHIFTS + 3;
       $)
    UNTIL
        SHIFTS = 18 DO NULL;
END = NUMB       " CONVERT_TO OCTAL " ;
```

```
216 CONVERT_TO_BINARY
217 CONVERT_TO_BINARY
218 CONVERT_TO_BINARY
219 CONVERT_TO_BINARY
220 CONVERT_TO_BINARY
221 CONVERT_TO_BINARY
222 CONVERT_TO_BINARY
223 CONVERT_TO_BINARY
224 CONVERT_TO_BINARY
225 CONVERT_TO_BINARY
226 CONVERT_TO_BINARY
227 CONVERT_TO_BINARY
228 CONVERT_TO_BINARY
229 CONVERT_TO_BINARY
230 CONVERT_TO_BINARY
231
232
233
234 CONV_TO_OCTAL
235 CONV_TO_OCTAL
236 CONV_TO_OCTAL
237 CONV_TO_OCTAL
238 CONV_TO_OCTAL
239 CONV_TO_OCTAL
240 CONV_TO_OCTAL
241 CONV_TO_OCTAL
242 CONV_TO_OCTAL
243 CONV_TO_OCTAL
244 CONV_TO_OCTAL
245 CONV_TO_OCTAL
246 CONV_TO_OCTAL
247 CONV_TO_OCTAL
248 CONV_TO_OCTAL
249 CONV_TO_OCTAL
250 CONV_TO_OCTAL
251 CONV_TO_OCTAL
252 CONV_TO_OCTAL
253 CONV_TO_OCTAL
254
255
```

```
PROCEDURE PROPAGATE_CARRY_IF_NECESSARY(BITS(WORD_SIZE) A;)

"  PROPAGATES CARRY DIGIT  "

    BITS(5) S;  BITS(24) Z;  BITS(4) OD;
    DEFINE REALNO : ACC2;
    IF A SHR 18 ¬= 0 THEN
     $(
         IF A & 7 > 4 THEN
             $(
                S:=3;
             OD:=A SHR 3 & 7 + 1;
             WHILE OD > 7 DO
                  $(
                       Z:=7 SHL S;  A:=A & ¬Z;
                         S:=S+3;  OD:=A SHR S & 7 + 1;
                    $)
                A:=A SHR 3;
                ACC2.CHAR:=ACC2.CHAR+1;
            $)
    $)
   ACC2.MANTISSA:=A;
END " PROPAGATE CARRY ";


PROCEDURE EQUALISE_CHARACTERISTICS();

   "  EQUALISES THE CHARACTERISTICS OF ACC AND ACC2 BEFORE
      OPERATION  "

   DEFINE REALNO : ACC,ACC2;

   IF ACC.CHAR > ACC2.CHAR THEN
     $(
        WHILE ACC.CHAR+ACC2.CHAR > 0 DO
          $(
              ACC2.MANTISSA:=ACC2.MANTISSA SHR 3;
              ACC2.CHAR:=ACC2.CHAR + 1;
          $)
     $)
```

```
                                    256
                                    257 PROPAGATE_CARRY_IF_NECE
                                    258 PROPAGATE_CARRY_IF_NECE
                                    259 PROPAGATE_CARRY_IF_NECE
                                    260 PROPAGATE_CARRY_IF_NECE
                                    261 PROPAGATE_CARRY_IF_NECE
                                    262 PROPAGATE_CARRY_IF_NECE
                                    263 PROPAGATE_CARRY_IF_NECE
                                    264 PROPAGATE_CARRY_IF_NECE
                                    265 PROPAGATE_CARRY_IF_NECE
                                    266 PROPAGATE_CARRY_IF_NECE
                                    267 PROPAGATE_CARRY_IF_NECE
                                    268 PROPAGATE_CARRY_IF_NECE
                                    269 PROPAGATE_CARRY_IF_NECE
                                    270 PROPAGATE_CARRY_IF_NECE
                                    271 PROPAGATE_CARRY_IF_NECE
                                    272 PROPAGATE_CARRY_IF_NECE
                                    273 PROPAGATE_CARRY_IF_NECE
                                    274 PROPAGATE_CARRY_IF_NECE
                                    275 PROPAGATE_CARRY_IF_NECE
                                    276 PROPAGATE_CARRY_IF_NECE
                                    277 PROPAGATE_CARRY_IF_NEC
                                    278 PROPAGATE_CARRY_IF_NEC
                                    279
                                    280
                                    281
                                    282 EQUALISE_CHARACTERISTI
AN ARITHMETIC                       283 EQUALISE_CHARACTERISTI
                                    284 EQUALISE_CHARACTERISTI
                                    285 EQUALISE_CHARACTERISTI
                                    286 EQUALISE_CHARACTERISTI
                                    287 EQUALISE_CHARACTERISTI
                                    288 EQUALISE_CHARACTERISTI
                                    289 EQUALISE_CHARACTERISTI
                                    290 EQUALISE_CHARACTERISTI
                                    291 EQUALISE_CHARACTERISTI
                                    292 EQUALISE_CHARACTERISTI
                                    293 EQUALISE_CHARACTERISTI
                                    294 EQUALISE_CHARACTERISTI
                                    295 EQUALISE_CHARACTERISTI
```

```
   ELSE
      WHILE ACC2.CHAR - ACC.CHAR > 0 DO
        $(
            ACC.MANTISSA:=ACC.MANTISSA SHR 3;
            ACC.CHAR:=ACC.CHAR+1;
        $)
END " EQUALISE_CHARACTERISTICS " ;


PROCEDURE TRUNCATE();

   DEFINE REALNO:ACC;
   BITS(4) DIGZ;
   IF ACC.CHAR <= 16 THEN
      ACC.MANTISSA:=0
   ELSE
     $(
         IF ACC.CHAR < 22 THEN
           $(
               DIGZ:=22-ACC.CHAR *3;
               ACC := ACC SHR DIGZ SHL DIGZ;
           $)
         ACC:=CONVERT_TO_BINARY(ACC);
     $)
END " TRUNCATE " ;


FUNCTION FLOAT(BITS(WORD_SIZE) INPUT;);

    "  FLAOTS AN INTEGER  "

    BITS(WORD_SIZE) OUTPUT;  BITS(3) DIGIT;  BITS(5) SHIFTS;
    FLAG NEGSIGN;
    DEFINE INT:INPUT;
    DEFINE REALNO : OUTPUT;
    IF INPUT.SIGN = 1 THEN
      $(
          INPUT.NUMB:=-INPUT.NUMB + 1;
          SET(NEGSIGN);
          INPUT.SIGN:=0;
```

```
32 |       $)                                                                  | 336 FLOAT
33 |                                                                           | 337 FLOAT
33 |     REPEAT                                                                | 338 FLOAT
34 |       $(                                                                  | 339 FLOAT
35 |            DIGIT:=INPUT REM 8;  INPUT:=INPUT/8;                           | 340 FLOAT
37 |            OUTPUT:=DIGIT SHL SHIFTS | OUTPUT;                           | 341 FLOAT
38 |            SHIFTS:=SHIFTS + 3;                                            | 342 FLOAT
39 |            IF SHIFTS > 15 THEN                                            | 343 FLOAT
40 |               $(                                                          | 344 FLOAT
41 |                  SET(VAL_TOO_BIG);  SET(ANY_INTERRUPT);                   | 345 FLOAT
43 |                   EXIT = 0;                                               | 346 FLOAT
44 |               $)                                                          | 347 FLOAT
45 |       $)                                                                  | 348 FLOAT
46 |       UNTIL INPUT = 0 DO NULL;                                            | 349 FLOAT
47 |     IF TRUE(NEGSIGN) THEN                                                 | 350 FLOAT
48 |       OUTPUT.SIGN:=1;                                                     | 351 FLOAT
49 |     OUTPUT.CHAR:=SHIFTS/3 + 16;                                           | 352 FLOAT
50 | END = OUTPUT;                                                             | 353 FLOAT
50 |                                                                           | 354
50 |                                                                           | 355
50 | PROCEDURE ADDREAL();                                                      | 356
52 |                                                                           | 357 ADDREAL
52 |     BITS(24) A;  DEFINE INT : A;  DEFINE REALNO : ACC,ACC2;               | 358 ADDREAL
55 |      FLAG NEGSIGN;                                                        | 359 ADDREAL
56 |     EQUALISE_CHARACTERISTICS();                                           | 360 ADDREAL
58 |     A:=CVB(ACC.SIGN,ACC.MANTISSA) + CVB(ACC2.SIGN,ACC2.MANTISSA);         | 361 ADDREAL
59 |     IF A.SIGN = 1 THEN                                                    | 362 ADDREAL
60 |       SET(NEGSIGN);                                                       | 363 ADDREAL
61 |     ACC2:=CONV_TO_OCTAL(A);                                               | 364 ADDREAL
62 |     PROPAGATE_CARRY_IF_NECESSARY(A);                                      | 365 ADDREAL
63 |     IF TRUE(NEGSIGN) THEN                                                 | 366 ADDREAL
64 |       ACC2.SIGN:=1;                                                       | 367 ADDREAL
65 |     UNSET(TOPFULL);                                                       | 368 ADDREAL
65 |                                                                           | 369 ADDREAL
65 | END " ADDREAL ";                                                          | 370 ADDREAL
65 |                                                                           | 371
65 | PROCEDURE SUBREAL();                                                      | 372
68 |                                                                           | 373 SUBREAL
68 |     BITS(24) A;  DEFINE REALNO : ACC,ACC2;  DEFINE INT : A;               | 374 SUBREAL
71 |     FLAG NEGSIGN;                                                        | 375 SUBREAL
```

```
    A:=CVB(ACC.SIGN,ACC.MANTISSA) + CVB(ACC2.SIGN,ACC2.MANTISSA);                    376 SUBREAL
    IF A.SIGN = 1 THEN                                                               377 SUBREAL
     SET(NEGSIGN);                                                                   378 SUBREAL
    ACC2:=CONV_TO_OCTAL(A);                                                          379 SUBREAL
    IF TRUE(NEGSIGN) THEN                                                            380 SUBREAL
      ACC2.SIGN:=1;                                                                  381 SUBREAL
    UNSET(TOPFULL);                                                                  382 SUBREAL
END " SUBREAL ";                                                                     383 SUBREAL
                                                                                     384
                                                                                     385
                                                                                     386
                                                                                     387
PROCEDURE INTARITH(BITS(4) OP;);                                                     388
                                                                                     389 INTARITH
    " PERFORMS CERTAIN INTEGER ARITHMETIC FUNCTIONS "                                390 INTARITH
                                                                                     391 INTARITH
    TEMPLATE INTEGER = SIGN(1),VALUE(23);                                            392 INTARITH
    DEFINE INTEGER : ACC,ACC2;                                                       393 INTARITH
    CASE OP OF                                                                       394 INTARITH
"ABI"   IF ACC.SIGN=1 THEN  ACC:= ¬ACC + 1¢1;                                        395 INTARITH
"SQI"   $(                                                                           396 INTARITH
            IF ACC.SIGN = 1 THEN ACC:= ¬ACC + 1;                                     397 INTARITH
            IF ACC.VALUE < LARGINT/ACC THEN                                          398 INTARITH
               ACC:=ACC*ACC                                                          399 INTARITH
            ELSE                                                                     400 INTARITH
               $(                                                                    401 INTARITH
                    SET(ANY_INTERRUPT); SET(VALUE_OUT_OF_RANGE);                     402 INTARITH
               $)                                                                    403 INTARITH
        $)                                                                           404 INTARITH
    ENDCASE;                                                                         405 INTARITH
                                                                                     406 INTARITH
END " INTARITH ";                                                                    407 INTARITH
                                                                                     408
PROCEDURE REALARITH(BITS(4) OP;);                                                    409
                                                                                     410 REALARITH
    " PERFORMS THE REAL ARITHMETIC OPERATIONS OF THE PASCAL MACHINE "                411 REALARITH
                                                                                     412 REALARITH
    TEMPLATE REAL = CHAR(5),SIGN(1),MANTISSA(18);                                    413 REALARITH
    DEFINE REAL : ACC,ACC2;                                                          414 REALARITH
    ACC:=NORMALISE(ACC);                                                             415 REALARITH
```

```
304 |     IF OP <= 4 THEN                                                                                                  |  416 REALARITH
305 |        ACC2:=NORMALISE(ACC2);                                                                                         |  417 REALARITH
306 |     CASE OP OF                                                                                                        |  418 REALARITH
307 | "ADR"    ADDREAL();                                                                                                   |  419 REALARITH
308 | "SBR"    SUBREAL();                                                                                                   |  420 REALARITH
309 | "MPR"    NULL;                                                                                                        |  421 REALARITH
310 | "DVR"    NULL;                                                                                                        |  422 REALARITH
311 | "NGR"    ACC.SIGN:=1;                                                                                                 |  423 REALARITH
312 | "FLT"    ACC:=FLOAT(ACC);                                                                                             |  424 REALARITH
313 | "FLO"    ACC2:=FLOAT(ACC2);                                                                                           |  425 REALARITH
314 | "TRC"    TRUNCATE();                                                                                                  |  426 REALARITH
315 | "ABR"    ACC.SIGN:=0;                                                                                                 |  427 REALARITH
316 |     ENDCASE;                                                                                                          |  428 REALARITH
317 | END " REALARITH ";                                                                                                    |  429 REALARITH
318 |                                                                                                                       |  430
319 |                                                                                                                       |  431
320 | FUNCTION BASE(BITS(6)LVL;);                                                                                           |  432
321 |                                                                                                                       |  433 BASE
322 |  " RETURNS THE BASE ADDRESS ON THE STACK OF THE DATA SEGMENT AT LEVEL LVL.                                            |  434 BASE
323 |    IT DOES THIS BY CHAINING DOWN THE APPROPRIATE NUMBER OF DATA SEGMENTS                                              |  435 BASE
324 |    TO FIND THE CORRECT ADDRESS.   "                                                                                  |  436 BASE
325 |                                                                                                                       |  437 BASE
326 |     BITS(WORD_SIZE) BADDR;     BADDR:=DATA_SEG_PNT;                                                                   |  438 BASE
327 |     WHILE LVL > 0 DO                                                                                                  |  439 BASE
328 |       $(                                                                                                              |  440 BASE
329 |           BADDR:=STORE(.BADDR+1.);                                                                                    |  441 BASE
330 |           LVL:=LVL-1;                                                                                                 |  442 BASE
331 |        $)                                                                                                             |  443 BASE
332 | END = BADDR;                                                                                                          |  444 BASE
333 |                                                                                                                       |  445
334 |                                                                                                                       |  446
335 | PROCEDURE EOF();                                                                                                      |  447
336 | END " EOF ";                                                                                                          |  448 EOF
337 |                                                                                                                       |  449
338 |                                                                                                                       |  450
339 | PROCEDURE STANDARD_PROCEDURE();                                                                                       |  451
340 | END " STANDARD_PROCEDURE " ;                                                                                          |  452 STANDARD_PROCEDURE
```

```
                                                                                    453
                                                                                    454
PROCEDURE MOV();                                                                    455
                                                                                    456 MOV
   BITS(WORD_SIZE) COUNT;                                                           457 MOV
   COUNT:=0;                                                                        458 MOV
   WHILE COUNT < Q DO                                                               459 MOV
     $(                                                                             460 MOV
         STORE(.ACC+COUNT.):=STORE(.ACC2+COUNT.);                                   461 MOV
         COUNT:=COUNT+1;                                                            462 MOV
     $)                                                                             463 MOV
END " MOV " ;                                                                       464 MOV
                                                                                    465
                                                                                    466
                                                                                    467
     PROCEDURE P_CODE_INTERPRETER();                                                468
                                                                                    469 P_CODE_INTERPRETER
     "  MAIN  INTERPRETATION LOOP    "                                              470 P_CODE_INTERPRETER
                                                                                    471 P_CODE_INTERPRETER
     TEMPLATE  PINST = OP(OP_SIZE),P(P_SIZE),Q(Q_SIZE);                             472 P_CODE_INTERPRETER
     DEFINE PINST:CINSTRREG;                                                        473 P_CODE_INTERPRETER
                                                                                    474 P_CODE_INTERPRETER
      CODEBASE(STORE);                                                              475 P_CODE_INTERPRETER
     WHILE FALSE(ENDPROG) DO                                                        476 P_CODE_INTERPRETER
       $(                                                                           477 P_CODE_INTERPRETER
           IF TRUE(ANY_INTERRUPT) THEN                                              478 P_CODE_INTERPRETER
              HANDLE_INTERRUPT();                                                   479 P_CODE_INTERPRETER
                                                                                    480 P_CODE_INTERPRETER
           CINSTRREG:=STORE(.PROGC.);                                               481 P_CODE_INTERPRETER
           PROGC:=PROGC + 1;                                                        482 P_CODE_INTERPRETER
           OP_CODE:=CINSTRREG.OP;              P:=CINSTRREG.P;                      483 P_CODE_INTERPRETER
           Q:=CINSTRREG.Q;                                                          484 P_CODE_INTERPRETER
                                                                                    485 P_CODE_INTERPRETER
           " IF INSTRUCTION NEEDS AT LEAST ONE OPERAND FROM THE STACK               486 P_CODE_INTERPRETER
             POP IT INTO ACC    "                                                   487 P_CODE_INTERPRETER
                                                                                    488 P_CODE_INTERPRETER
           IF OP_CODE < 37 THEN                                                     489 P_CODE_INTERPRETER
              ACC:=POP();                                                           490 P_CODE_INTERPRETER
                                                                                    491 P_CODE_INTERPRETER
           " SIMILIARY IF INSTRUCTION NEEDS ANOTHER OPERAND  POP IT                 492 P_CODE_INTERPRETER
             INTO ACC2.  THIS ENSURES EVERYTHING IS IN A REGISTER                   493 P_CODE_INTERPRETER
             HENCE INCREASING EFFICIENCY   "                                        494 P_CODE_INTERPRETER
                                                                                    495 P_CODE_INTERPRETER
```

```
59 |            IF  OP_CODE < 24 THEN                                                  |  496 P_CODE_INTERPRETER
60 |               ACC2:= POP();                                                       |  497 P_CODE_INTERPRETER
61 |                                                                                   |  498 P_CODE_INTERPRETER
61 |            IF OP_CODE  < 37 THEN                                                  |  499 P_CODE_INTERPRETER
62 |              $(                                                                   |  500 P_CODE_INTERPRETER
63 |                   CASE OP_CODE OF                                                 |  501 P_CODE_INTERPRETER
64 |                                                                                   |  502 P_CODE_INTERPRETER
64 |                      " LOGICAL  OPERATIONS "                                      |  503 P_CODE_INTERPRETER
64 |                                                                                   |  504 P_CODE_INTERPRETER
64 | "AND"                ACC:= ACC & ACC2;                                            |  505 P_CODE_INTERPRETER
65 | "IOR"                ACC:= ACC | ACC2;                                           |  506 P_CODE_INTERPRETER
65 |                                                                                   |  507 P_CODE_INTERPRETER
65 |                      " SET OPERATIONS  "                                          |  508 P_CODE_INTERPRETER
65 |                                                                                   |  509 P_CODE_INTERPRETER
65 | "DIF"                ACC:=ACC XOR ACC2;                                           |  510 P_CODE_INTERPRETER
67 | "INT"                ACC:=ACC & ACC2;                                             |  511 P_CODE_INTERPRETER
68 | "INN"                IF 1 SHR ACC2 & ACC = 0 THEN                                 |  512 P_CODE_INTERPRETER
69 |                         ACC:=LTRUE                                                |  513 P_CODE_INTERPRETER
70 |                      ELSE                                                         |  514 P_CODE_INTERPRETER
71 |                        ACC:=LFALSE;                                               |  515 P_CODE_INTERPRETER
71 | "UNI"                ACC:=ACC | ACC2;                                            |  516 P_CODE_INTERPRETER
72 |                                                                                   |  517 P_CODE_INTERPRETER
72 |                      " INTEGER OPERATIONS "                                       |  518 P_CODE_INTERPRETER
72 |                                                                                   |  519 P_CODE_INTERPRETER
72 | "ADI"                ACC:=ACC + ACC2;                                             |  520 P_CODE_INTERPRETER
73 | "SBI"                ACC:=ACC - ACC2;                                             |  521 P_CODE_INTERPRETER
74 | "MPI"                ACC:=ACC * ACC2;                                             |  522 P_CODE_INTERPRETER
75 | "MOD"                ACC:=ACC REM ACC2;                                           |  523 P_CODE_INTERPRETER
75 | "DVI"                ACC:=ACC / ACC2;                                             |  524 P_CODE_INTERPRETER
77 |                                                                                   |  525 P_CODE_INTERPRETER
77 |                      " REAL OPERATIONS "                                          |  526 P_CODE_INTERPRETER
77 | "ADR"                REALARITH(1);                                                |  527 P_CODE_INTERPRETER
78 | "SBR"                REALARITH(2);                                                |  528 P_CODE_INTERPRETER
79 | "MPR"                REALARITH(3);                                                |  529 P_CODE_INTERPRETER
80 | "DVR"                REALARITH(4);                                                |  530 P_CODE_INTERPRETER
81 |                                                                                   |  531 P_CODE_INTERPRETER
81 |                      " CONDITIONAL OPERATIONS "                                   |  532 P_CODE_INTERPRETER
81 |                                                                                   |  533 P_CODE_INTERPRETER
81 | "EQU"                TEST(1);                                                     |  534 P_CODE_INTERPRETER
82 | "NEQ"                 $( TEST(1);          ACC:=¬ACC;      $)                    |  535 P_CODE_INTERPRETER
```

```
386 |  "GEQ"                TEST(2);                                                  | 536 P_CODE_INTERPRETER
387 |  "GRT"                TEST(3);                                                  | 537 P_CODE_INTERPRETER
388 |  "LEQ"               $(   TEST(3);          ACC:=¬ACC;     $)                 | 538 P_CODE_INTERPRETER
392 |  "LES"               $(   TEST(2);          ACC:= ¬ACC;     $)                | 539 P_CODE_INTERPRETER
395 |                                                                               | 540 P_CODE_INTERPRETER
395 |                      " MISCELLANEOUS INSTRUCTIONS "                           | 541 P_CODE_INTERPRETER
396 |                                                                               | 542 P_CODE_INTERPRETER
396 |  "MOV"               MOV();                                                   | 543 P_CODE_INTERPRETER
397 |  "IXA"               ACC:=Q * ACC + ACC2;                                     | 544 P_CODE_INTERPRETER
398 |                                                                               | 545 P_CODE_INTERPRETER
398 |                      " NOW ONTO INSTRUCTIONS WHICH ONLY OPERATE ON            | 546 P_CODE_INTERPRETER
398 |                        THE ELEMENT ON TOP OF THE STACK(HELD IN ACC)  "        | 547 P_CODE_INTERPRETER
398 |  "NGI"               ACC:= ¬ACC + 1;                                          | 548 P_CODE_INTERPRETER
399 |  "NGR"               REALARITH(5);                                            | 549 P_CODE_INTERPRETER
400 |  "NOT"               ACC:=¬ACC;                                               | 550 P_CODE_INTERPRETER
401 |  "FLT"               REALARITH(6);                                            | 551 P_CODE_INTERPRETER
402 |  "FLO"               REALARITH(7);                                            | 552 P_CODE_INTERPRETER
403 |  "TRC"               REALARITH(8);                                            | 553 P_CODE_INTERPRETER
404 |  "ABI"               INTARITH(1);                                             | 554 P_CODE_INTERPRETER
405 |  "ABR"               REALARITH(9);                                            | 555 P_CODE_INTERPRETER
406 |  "INC"               ACC:=ACC + Q;                                            | 556 P_CODE_INTERPRETER
407 |  "SGS"               ACC:=1 SHR ACC;                                          | 557 P_CODE_INTERPRETER
408 |  "SQI"               INTARITH(2);                                             | 558 P_CODE_INTERPRETER
409 |  "SQR"               REALARITH(10);                                           | 559 P_CODE_INTERPRETER
410 |  "ODD"               ACC:=ACC REM 2;                                          | 560 P_CODE_INTERPRETER
411 |  "DEC"               ACC:=ACC - Q;                                            | 561 P_CODE_INTERPRETER
412 |                    ENDCASE;                                                   | 562 P_CODE_INTERPRETER
412 |                                                                               | 563 P_CODE_INTERPRETER
412 |                      " NOW PUSH ACC ONTO STACK "                              | 564 P_CODE_INTERPRETER
412 |                                                                               | 565 P_CODE_INTERPRETER
412 |                      PUSH();                                                  | 566 P_CODE_INTERPRETER
413 |                 $)                                                            | 567 P_CODE_INTERPRETER
414 |               ELSE                                                            | 568 P_CODE_INTERPRETER
414 |                  "  NOW DEAL WITH VARIOUS INSTRUCTIONS WHICH MAY OR          | 569 P_CODE_INTERPRETER
414 |                     MAY NOT OPERATE ON THE STACK "                            | 570 P_CODE_INTERPRETER
414 |                                                                               | 571 P_CODE_INTERPRETER
414 |                   $(                                                          | 572 P_CODE_INTERPRETER
415 |                    CASE OP_CODE - 37  OF                                      | 573 P_CODE_INTERPRETER
416 |                                                                               | 574 P_CODE_INTERPRETER
    |                      " LOAD AND STORE INSTRUCTIONS "                          | 575 P_CODE_INTERPRETER
```

```
                                                                                                                    576 P_CODE_INTERPRETER
                                                                                                                    577 P_CODE_INTERPRETER
"LOD"               $(                                                                                              578 P_CODE_INTERPRETER
                        ACC:= STORE(. BASE(P) + Q .);                                                               579 P_CODE_INTERPRETER
                        IF ACC = UNDEF THEN                                                                         580 P_CODE_INTERPRETER
                           $(                                                                                       581 P_CODE_INTERPRETER
                               SET(ANY_INTERRUPT); SET(UNINITIALISED_VALUE);                                        582 P_CODE_INTERPRETER
                           $)                                                                                       583 P_CODE_INTERPRETER
                        ELSE                                                                                        584 P_CODE_INTERPRETER
                           PUSH();                                                                                  585 P_CODE_INTERPRETER
                    $)                                                                                              586 P_CODE_INTERPRETER
                                                                                                                    587 P_CODE_INTERPRETER
"LOD"               $(                                                                                              588 P_CODE_INTERPRETER
                       ACC:= STORE(.Q.);                                                                            589 P_CODE_INTERPRETER
                       IF ACC = UNDEF THEN                                                                          590 P_CODE_INTERPRETER
                         $(                                                                                         591 P_CODE_INTERPRETER
                            SET(ANY_INTERRUPT); SET(UNINITIALISED_VALUE);                                           592 P_CODE_INTERPRETER
                         $)                                                                                         593 P_CODE_INTERPRETER
                       ELSE                                                                                         594 P_CODE_INTERPRETER
                          PUSH();                                                                                   595 P_CODE_INTERPRETER
                    $)                                                                                              596 P_CODE_INTERPRETER
                                                                                                                    597 P_CODE_INTERPRETER
"LAD"               $(                                                                                              598 P_CODE_INTERPRETER
                        ACC:=BASE(P) + Q;          PUSH();                                                          599 P_CODE_INTERPRETER
                    $)                                                                                              600 P_CODE_INTERPRETER
"LAO"               $(                                                                                              601 P_CODE_INTERPRETER
                        ACC:= Q;       PUSH();                                                                      602 P_CODE_INTERPRETER
                    $)                                                                                              603 P_CODE_INTERPRETER
"LDC"                                                                                                               604 P_CODE_INTERPRETER
  "LDCI"              $(                                                                                            605 P_CODE_INTERPRETER
                        ACC:=STORE(.Q.);          PUSH();                                                           606 P_CODE_INTERPRETER
                    $)                                                                                              607 P_CODE_INTERPRETER
"LCA"               $(                                                                                              608 P_CODE_INTERPRETER
                        ACC:=Q;         PUSH();                                                                     609 P_CODE_INTERPRETER
                    $)                                                                                              610 P_CODE_INTERPRETER
"STR"               STORE(. BASE(P) + Q .) := POP();                                                                611 P_CODE_INTERPRETER
"SRO"               STORE(.Q.):=POP();                                                                              612 P_CODE_INTERPRETER
"STO"               $(                                                                                              613 P_CODE_INTERPRETER
                        ACC:=POP();       ACC2:=POP();                                                              614 P_CODE_INTERPRETER
                    $)                                                                                              615 P_CODE_INTERPRETER
```

```
                "  OTHER MISCELLANEOUS  MACHINE INSTRUCTIONS "
"IND"           $(
                    ACC:=POP() + Q;
                    IF STORE(.ACC.) = UNDEF THEN
                       $(
                            SET(ANY_INTERRUPT);
                            SET(UNINITIALISED_VALUE);
                       $)
                    ELSE
                       $(   ACC:=STORE(.ACC.);     PUSH();    $)
                $)

"MST"           $(
                ACC:=UNDEF;         PUSH();
                ACC:=BASE(P);       PUSH();
                ACC:=DATA_SEG_PNT;        PUSH();
                ACC:=UNDEF;         PUSH();
                $)
"CUP"           $(
                DATA_SEG_PNT:=STACKP - P - 3;
                STORE(.DATA_SEG_PNT + 3.):=PROGC;
                PROGC:=Q;
                $)
"ENT"           $(
                ACC2:=DATA_SEG_PNT + Q;
                IF ACC2 > HEAPP THEN
                   $(
                        SET(ANY_INTERRUPT);  SET(STACK_OFLOW);
                   $)
                IF STACKP < INPUTADR THEN
                   STACKP:=PRDADR;
                ACC:=STACKP + 1;
                WHILE ACC <= ACC2 DO
                   STORE(.ACC.):=UNDEF;
                STACKP:=ACC2;
                $)
"RET"           $(
                IF P = 0 THEN
                   STACKP:=DATA_SEG_PNT - 1
```

```
616 P_CODE_INTERPRETER
617 P_CODE_INTERPRETER
618 P_CODE_INTERPRETER
619 P_CODE_INTERPRETER
620 P_CODE_INTERPRETER
621 P_CODE_INTERPRETER
622 P_CODE_INTERPRETER
623 P_CODE_INTERPRETER
624 P_CODE_INTERPRETER
625 P_CODE_INTERPRETER
626 P_CODE_INTERPRETER
627 P_CODE_INTERPRETER
628 P_CODE_INTERPRETER
629 P_CODE_INTERPRETER
630 P_CODE_INTERPRETER
631 P_CODE_INTERPRETER
632 P_CODE_INTERPRETER
633 P_CODE_INTERPRETER
634 P_CODE_INTERPRETER
635 P_CODE_INTERPRETER
636 P_CODE_INTERPRETER
637 P_CODE_INTERPRETER
638 P_CODE_INTERPRETER
639 P_CODE_INTERPRETER
640 P_CODE_INTERPRETER
641 P_CODE_INTERPRETER
642 P_CODE_INTERPRETER
643 P_CODE_INTERPRETER
644 P_CODE_INTERPRETER
645 P_CODE_INTERPRETER
646 P_CODE_INTERPRETER
647 P_CODE_INTERPRETER
648 P_CODE_INTERPRETER
649 P_CODE_INTERPRETER
650 P_CODE_INTERPRETER
651 P_CODE_INTERPRETER
652 P_CODE_INTERPRETER
653 P_CODE_INTERPRETER
654 P_CODE_INTERPRETER
655 P_CODE_INTERPRETER
```

```
500 |                        ELSE                                                                                    |  656 P_CODE_INTERPRETER
500 |                           STACKP:=DATA_SEG_PNT;                                                                |  657 P_CODE_INTERPRETER
501 |                        PROGC:=STORE(.DATA_SEG_PNT + 3.);                                                       |  658 P_CODE_INTERPRETER
502 |                        DATA_SEG_PNT:=STORE(.DATA_SEG_PNT + 2.);                                                |  659 P_CODE_INTERPRETER
503 |                     $)                                                                                         |  660 P_CODE_INTERPRETER
504 |  "CSP"              STANDARD_PROCEDURE();                                                                      |  661 P_CODE_INTERPRETER
505 |  "CHK"              IF STORE(.STACKP.) < STORE(.Q-1.) OR                                                       |  662 P_CODE_INTERPRETER
506 |                        STORE(.STACKP.) > STORE(.Q.) THEN                                                       |  663 P_CODE_INTERPRETER
506 |                          $(                                                                                    |  664 P_CODE_INTERPRETER
507 |                             SET(ANY_INTERRUPT);                                                                |  665 P_CODE_INTERPRETER
508 |                             SET(VALUE_OUT_OF_RANGE);                                                           |  666 P_CODE_INTERPRETER
509 |                          $)                                                                                    |  667 P_CODE_INTERPRETER
510 |  "EOF"              EOF();                                                                                     |  668 P_CODE_INTERPRETER
511 |  "UJP"              PROGC:=Q;                                                                                  |  669 P_CODE_INTERPRETER
512 |  "XJP"              PROGC:=POP() + Q;                                                                          |  670 P_CODE_INTERPRETER
513 |  "FJP"              IF POP() = LFALSE THEN                                                                     |  671 P_CODE_INTERPRETER
514 |                        PROGC:=Q;                                                                               |  672 P_CODE_INTERPRETER
515 |  "STP"              SET(ENDPROG);                                                                              |  673 P_CODE_INTERPRETER
516 |                  ENDCASE;                                                                                      |  674 P_CODE_INTERPRETER
516 |                    $)                                                                                          |  675 P_CODE_INTERPRETER
517 |                                                                                                                |  676 P_CODE_INTERPRETER
517 |           $)                                                                                                   |  677 P_CODE_INTERPRETER
518 |   END " P_CODE_INTERPRETER " ;                                                                                 |  678 P_CODE_INTERPRETER
519 |                                                                                                                |  679
519 |        " ********** "    P_CODE_INTERPRETER();  " ********** "                                                |  680
520 |  ENDPROGRAM;                                                                                                   |  681
```

****

COMPILATION COMPLETE -- NO OF ERRORS = 0

1685 MICROINSTRUCTIONS WERE GENERATED

APPENDIX 4

AN ALGORITHM DESCRIPTION LANGUAGE

HERIOT-WATT UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

D.S. 5/76/5

THE DESCRIPTION OF ALGORITHMS

I. SOMMERVILLE

OCTOBER, 1976.

The material in this appendix is made up of four program listings:-

(1) A listing of the SUILVEN code plus generated microcode for a simple s-machine called SIMPS. This is included to illustrate the format of the output produced by the SUILVEN compiler.

(2) A listing of the SUILVEN code implementing the SASL s-machine.

(3) A listing of the output produced by the B1700 simulator when executing a SASL program to sum the elements of a list.

(4) A listing of the SUILVEN code implementing the PASCAL s-machine.

The code given here and the examples given in the body of the thesis may not exactly correspond. This is due to the fact that both the SASL and the PASCAL machines were re-implemented and opportunity was taken to improve them. The re-implementation was necessary because the author of the thesis left St Andrews University and had no access to the machine there. The programs which were written were supposedly portable but, as usual, this portabilty turned out to be mythical and an inordinate amount of effort was involved in transporting the various programs.

The PASCAL machine implementation was only developed to the stage where the salient features of SUILVEN are illustrated and not to the stage where PASCAL programs actually run on the machine. As we had decided to abandon the development of SUILVEN, we did not feel that the effort of completely implementing the PASCAL machine was justified.

APPENDIX 3

EXAMPLES

1. INTRODUCTION

This document lays down a set of guidelines (rather than a rigid notation) for the description of algorithms. A language for describing algorithms is presented along with notes on the format of an algorithms description. Some examples illustrate the use of the language.

The notation described below has been adopted as the standard algorithm description notation of the Department of Computer Science, Heriot-Watt University. It should be used by all staff and students who produce descriptions of algorithms.

2. ALGORITHM DESCRIPTION

The production of algorithm descriptions is necessary at two stages in the development of a software system

(i) Before the program is written to communicate the method of problem solving to the programmer.

(ii) After the program has been written to communicate the problem solving techniques actually used.

If the programmer participates at the design stage of a program a personal, informal algorithm description may be used. However, the language described below should be used whenever it is necessary to communicate an algorithm design to someone else. This is necessary when several individuals are working as a programming team and when a completed program is documented.

In describing the algorithms used in a computer program a conflict arises. The higher-level an algorithm description the more readable and understandable it is. However, as the description progresses to higher and higher levels the correspondence between the description and the program text becomes more nebulous. It is necessary to establish some kind of compromise which is at a much higher level than a programming language, yet still exhibits typical program features such as sequencing.

The language described in section 3 allows a measure of flexibility in algorithm description. Although sequencing is defined, operations may be expressed formally or informally as programming language statements or as English text.

3. A LANGUAGE FOR THE DESCRIPTION OF ALGORITHMS

The language description below is expressed in a slightly extended BNF. The enclosure of an element in starred square brackets [ ]* indicates that that element may be repeated zero or more times. Hopefully the language semantics are fairly obvious from the syntactic description.

<algorithm description> ::= <algorithm name> <body> END <algorithm name>

<body> ::= <statement> [<statement>]*

<statement> ::= <If>|<While>|<Do>|<Select>|<name>|<text>|<compound>

<If> ::= <If clause> <statement>|<If clause> <statement> ELSE <statement>

<If clause> ::= IF <condition> THEN

<While> ::= WHILE <condition> DO <statement>

<Do> ::= DO <statement> UNTIL <condition>

<Select> ::= SELECT <guarded statement list> ENDSELECT ELSE <statement>

<guarded statement list> ::= <guarded statement> [<guarded statement>]*

<guarded statement> ::= <condition> : <statement>

<compound> ::= {<body>}

<name> ::= <identifier>

<condition> ::= <boolean expression>|<text>

<text> ::= *Any English phrase or sentence*

Any text preceded by a % is regarded as a comment.

The above statements should be familiar apart from, possibly, the select statement. The example in section 4 illustrates the use of the select statement. Informal English descriptions may be used as a condition or as a statement where this is appropriate.

4. THE FORMAT OF ALGORITHM DESCRIPTIONS

It is desirable that algorithm descriptions should comply with a fairly rigid format. Some suggestions as to what this format should be are set out below.

(i) The algorithm name should be on a line by itself as should the END declaration

(ii) The algorithm body should be indented within the name and the END declaration.

5. EXAMPLES

The examples below describe typical algorithms which might be used in a top-down recursive descent parser for an ALGOL-like programming language.

```
IFCOMMAND
        % Parses if statements
        NEXTSYMBOL
        CONDITION
        IF Symbol ≠ "THEN" THEN
            ERROR ("Then necessary")
        ELSE
            {
             NEXTSYMBOL
             IF Symbol = "IF" THEN
                ERROR ("If forbidden here")
             ELSE
                STATEMENT
             IF Symbol = "ELSE" THEN
                {
                 NEXTSYMBOL
                 STATEMENT
                }
            }
END IFCOMMAND
STATEMENT
        SELECT
            Symbol = "IF":IFCOMMAND
            Symbol = "WHILE":WHILECOMMAND
                      :
            Symbol = identifier:{
                                  Name := Symbol
                                  NEXTSYMBOL
                                  IF Symbol = "(" THEN
                                     PROCEDURECALL (Name)
                                  ELSE
                                     ASSIGNMENT (Name)
                                 }
        ENDSELECT
            ELSE
                ERROR ("Bad symbol")
        NEXTSYMBOL
END STATEMENT
```

(iii) Each statement should begin on a separate line.

(iv) Statements which are part of a control statement should be indented.

(v) Names of other algorithms should be in block capitals, names of variables should be lower-case letters or script and reserved words should be underlined.

(vi) When describing all the algorithms in a program, the main program should be described first followed by the algorithms of the procedures which it calls etc. etc.

(vii) The curly brackets {}, may be replaced by any other pair of bracketing symbols at the discretion of the user.

Notice the use of the select statement in this example. The condition for selecting a statement for execution must be explicitly stated. This can be translated either into an ALGOL case statement or into nested IF statements.