MACHINE ARCHITECTURE AND THE PROGRAMMING LANGUAGE BCPL

by

MARK C. FOX

B.Sc.

The University of British Columbia, 1975

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard.

THE UNIVERSITY OF BRITISH COLUMBIA

September, 1978

Department of Computer Science

The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1W5

Date Sept, 12. 1978.

0. Abstract

This thesis describes the design of a well mapped machine [1] for the language BCPL. Based on a generalized notion of stack machines the SLIM (Stack Language for Intermediate Machines) machine is described. As the acronym suggests, representation of BCPL programs in SLIM is in fact slim compared with other architectures. The utility of this measure for comparison with other architectures is discussed and some encouraging results presented. Apart from this result, some advance is made in the classical mode of porting BCPL programs. Normally the compiler produces OCODE from which INTCODE is generated. The BCPL SLIM compiler shortcuts this process by generating SLIM directly from the program tree thus dispensing with software corresponding to the OCODE to INTCODE translator. Translation of BCPL programs is thus simplified and speeded up.

[1] by well mapped we mean that transformations in the high level language correspond closely to those in the low level machine representation.

List of tables

List of figures

Acknowledgements

I would like thank the Department for the provision of various Teaching assistantships. My thanks also go to my supervisor Sam Chanson, for his provision of a research assistantship that allowed me to devote more time to this work, and also for his constructive criticism of my thesis. Harvey Abramson is also to be thanked for reading this thesis.

Outside the department I am grateful for other friends and interests at UBC particularly the community of Fairmont House that were there during the academic frustrations of this past year. For the broader perspective provided by these people I am very thankful.

1. Introduction

As with most work this thesis needs to be set in its proper context. The historical perspective is one aspect of this but more importantly there are a number of current issues that give this thesis relevance. We will first examine these two components of this thesis's context and then proceed to outline the objectives that governed the realization of the SLIM machine.

1.1 Setting the context

The way programming languages are used is of interest to various people. These people often include systems architects, language designers and compiler writers. In the design of a programming language it is useful to know the kind of constructs that are most frequently used. Compiler writers can use this knowledge effectively as they decide how much energy to devote to compiling good code for the more common constructs. By good code we mean code that compactly and efficiently represents the intentions of the language constructs used. For example, the dominance of the assignment statement in programs is one candidate for which good code should be compiled. System architects are more interested in how effectively the language maps to the machine and empirical evaluation can lead to some fine tuned application oriented architectures.(see [1])

A number of people have studied the ways in which programming languages are used: Tannenbaum [2] has studied a BCPL variant called SAL and proposed a simple machine architecture ; Knuth [3] has analyzed Fortran programs; Alexander [4] studied XPL as implemented on the IBM and unearthed some inconsistency in the mapping from XPL to IBM assembler. Wortman [5] studied student PL on an interpretative system and used his analysis to incorporate changes into his PL machine. He states:"We took as our design goal the development of new design tools to aid the designer in building computers that satisfied the actual rather than the imagined needs of the programmer." Gordon and Capstick [6] have examined COBOL. Of course we would be amiss if we failed to mention the design of the Burrough's machines which were high level language machines in the first place. An analysis of over 60 ALGOL 60 compilers by Wichmann [7] showed code produced by the Burrough's compiler occupied <u>half the space of code produced by the IBM compiler</u>. This level of approaching the problem from the language point of view has gone hand in hand with the development of microprogramming.

Once the manufacturers offered user microprogramming there was a flood of activity in this field. Many machines were designed which were truly meant to be very general purpose (see [8] and [9] for examples) . The Nanodata QM-1 was perhaps too flexible but the emulation of the PDP-11 on it [10] proved it to be useful. As usual, Burroughs in the design of the B1700 series (see [11]) seriously and effectively attempted to free us from von Neumann style machines. Wilner states that "Von Neumann derived machines are automatous malefactors who force programmers to lie on many procrustean beds". Each particular language would have its own S (S standing for secondary) machine optimized for its own particular application area that would be emulated by B1700 hardware. No machine language was built into the hardware and therefore each language to be executed had to first reconfigure the B1700 processor. Concurrent execution of S machines was very feasible with the fast switching time (14 - 53 microsec) . Apart from this architectural concern on the level of machine design, microprogramming itself was used to measure computer systems. Saal [12] used this very transparent technique to obtain system design guidelines.

Despite all these activities that brought architecture to the fore as a research area, Rosin who was involved for a long time with microprogramming, was forced to define microprogramming as "the implementation of hopefully reasonable systems through interpretation on unreasonable machines" [13]. Even this pessimistic comment should not detract from the overriding concern with machine architecture not just in itself as an end but as a means to facilitating what people want to do. It is in this light that we should see the development of SLIM.

1.2 Current issues

Computer science with its concern for easy and effective expression has long been in the business of generating new languages. Translators consequently tend to be one of the most frequently used pieces of software and will probably continue to stay that way. Translator writing systems are a manifestation of this fact. However as Appelbe [14] states: "The major complexities encountered in the design and implementation are usually in the 'semantics phase' of the translator in generating the object program from an internal source program representation". This complexity no doubt arises from a number of sources: Languages that are exceedingly complex and hence inevitably require complex code generation. Another more important source of complexity is that inhospitable host architectures demand contortions on the part of the code generator and hence the implementor. This is the complete reversal of the situation in the parsing-syntax analysis phase where the methods are very well understood. Despite this acknowledgement, computer science has tended to minimize the importance of machine architecture and has often comforted itself with the fact that the language was implemented and left it at that. The implementation was the overriding concern and after all, with cheaper memory prices we are not really concerned about how efficiently our programs are represented, are we? This hides the main point. It's high time that computer science not relegate machine design only to the military and the artificial intelligence communities where an overwhelming need demands better architecture. We need to refute the notion that the description of a machine's assembly language constitutes its complete definition.

At present, even to the most casual observer, there is an explosion in the area of microprocessor technology. The market here is in a continual state of flux as more and more products are announced. The availability of bit slice components makes possible the construction of new machines at reasonable costs. T.M. McWilliams et al. [15] describe the construction of a PDP/11 using bit slices for a total cost of $1076. This machine even outperforms the LSI-11. With this continual state of flux industry has adopted a microprocessor chip standard - the Intel 8080. In light of the above fact this is not the most desirable chip from an architectural point of view. Perhaps this standardization was unavoidable. Another more important standardization of this technology is the language and its implementation. BASIC has become the standard language with a variety of implementations. Its implementation however is more significant as far as SLIM is concerned. Interpretation is the accepted way of implementing Basic. From a very rudimentary analysis of the BASIC source these interpreters interpret BASIC programs at a fairly high level.

Before we draw out the significance of these facts we shall quote from the Nov 15, 1977 draft of the objectives for computer science in the department of Computer Science at UBC [16]. "Broadly speaking computer science is concerned with the design of algorithms and with efficient implementations of algorithms on computing systems. The computing systems may vary in size from the hand held programmable calculator to a complex collection of devices interconnected by satellite and cable."

Let us examine this statement in the context of high school computing facilities. The systems that high school students are dealing with are Basic ones in more than one sense! As a department there are educational issues at stake. What are potential university students in computer science being exposed to? Is the form of expression really suited to developing structured programming? Will university programs at the first year level involve a certain amount of deprogramming? One attempt to address this issue of teaching computer science as a unified discipline was Peck's 'Essence of computer science' [17]. In this report the language BCPL and its abstract machine INTCODE served as a reference point for teaching computer science coherently. In order to execute BCPL programs one interpreted their INTCODE representation. The major limitation of the INTCODE system was the size of the INTCODE version of the compiler since if the compiler cannot fit on the host system, it becomes very cumbersome to compile programs on one machine and execute them (via interpretation) on another. SLIM serves exactly the same function as INTCODE except that it is much more compact (as we show in Chapter 4) and hence the BCPL SLIM compiler is more compact than the INTCODE version . However both forms of realizing BCPL are exactly similar to the current form of realizing BASIC - interpretation. Therefore if computer science is concerned with the representation of algorithms and BCPL is accepted as a valid vehicle for this, then if SLIM facilitates this process on a wide variety of machines it should be of concern and use to the department in realizing one of its stated objectives. There seems to be the very real possibility that computer science could be making inroads here not only in the realization of more effective languages but also in the realization of more effective hardware as in the bit slice version of the PDP/11.

One final issue involves the language BCPL (see [20], [21] and [22]) . We will just present the case for this class of language. By class we mean systems programming languages of the BCPL type. OS-6 [19], a single user operating system was written almost completely in BCPL. C, a BCPL offshoot with PDP/11 constructions that have filtered up into the language, is the source langauge for a very effective operating system UNIX, [23] written for the PDP/11. This demonstrates the utility and effectiveness of this class of language. Therefore our concern with it is not misplaced.

A more local BCPL issue concerns the classical form of translation to INTCODE. BCPL source is first translated to OCODE and from this one translates to INTCODE. SLIM is generated directly from the tree representation of the BCPL program. A number of advantages accrue:

i. One complex piece of software (OCODE to INTCODE translator) is dispensed with.

ii. The obscure OCODE machine no longer confronts us.

iii. The realization that translation from the tree to SLIM is straightforward and that simple optimizations are easily handled.

For too long we have been stuck with the BCPL-> OCODE -> INTCODE process. This thesis shows that this procedure is no longer necessary, and more importantly shows that these two virtual machines are not ideally suited to BCPL.

1.3 Objectives

The prime objective was to achieve compact program representation. By compact program representation we mean that the size of the translated program is small. This is always a convenient result but more importantly it reflects how effective the mapping is from language to machine. The choice of this measure rather than time for example, will be dealt with later when we address the issue of how one actually evaluates an architecture. Presently there is no standard evaluation technique that allows for program independent evaluation of an architecture.

Simplicity was the second criteria. Partly this is a reaction against the trend that dictates complexity to be the norm, but a simple architecture has a number of advantages. Simple architectures are more easily understood and can exemplify architectural principles. If interpretation is going to be the sole method of executing BCPL programs then the simpler the machine to be emulated the simpler the emulator. As this most probably will be the form of execution in the microprocessor field and since the emulator will most probably be written in assembler, the ease with which the interpreter is implemented is very important. If we extend the notion of interpretation and temporarily equate it with microprogramming, then the same advantages apply except that in this case the corresponding assembly language is more primitive. If one actually intends to glue together the SLIM machine using bit slice technology then the simpler the machine the better.

A third guideline was that the machine acknowledge the existence of BCPL right at the machine level. Thus in effect the SLIM machine is as easily described by BCPL as it is by the SLIM assembly language. For example the CALL instruction should actually do more than just change the program counter and remember its previous value. The IBM 370 BALR instruction is a classic example of refusing to acknowledge that a programmer's common forms of expression are at a much higher level and that changing control is much more than a change in location. This guideline hopes to demonstrate that the translation process can be straightforward and not always involve numerous contortions in the code generation section of the translator.

With these objectives in mind, and the context in which this thesis is set outlined, we now proceed to describe the SLIM machine in an informal manner.

2. The SLIM machine description

This machine is offered as an alternative to the current way of porting BCPL programs. This section contains a specification of the machine but offers no justification or motivation for the choice of SLIM. However as the acronym suggests (a Stack Language for Intermediate Machines) , one advantage should be that of compact program representation or SLIM code.

2.1 Preliminaries

The memory of the SLIM machine like INTCODE [22] and PICA-B [26] , consists of an array of cells numbered from zero. The consecutive numbers assigned to cells are known as their addresses. The number of bits per cell is unspecified.

The SLIM machine has five registers. One accumulator (ACC) is used for all arithmetic, logical and various other operations. P is used as an index register and points to the base of the current stack frame and S points to the top of the current stack frame. (Figure 2.1) The C register is used as a program counter. G is used to access the base of the location of the global variables. We will justify this choice later, but for the present we will give some examples of equivalent constructions in present day architectures. The HP-21MX's base page addressing functions in very much the same way. Even the PDP-8's zeroth page addressing, in which every page can access the zeroth page, is a form of global variable access. Two more respected architectures- the HP 3000 and the B5500 (see [24]) have similar features. The DB register in the HP3000 points to the base of the global area that is in fact kept at the bottom of the stack. The 5500's R register points to the separate (i.e., not in the stack) program reference table that contains global variables and global procedure names.

To use the current terminology the SLIM machine is a single accumulator, one address machine.

```
Previous stack frames      Current stack frame


┌────────┬────────┬───────────┬──────────┬──────┐
|        |Linkage |Parameters|  Local    | .... |
|        |  Info  |          |variables  |      |
└────────┴────────┴───────────┴──────────┴──────┘
         ^                                      ^
         |                                      |
         |                                      |
         |                                      |
         P                                      S
```

Figure 2.1 The runtime stack

2.2 Variables

Before describing the operations provided by the SLIM machine we will look at the four sets of variables in BCPL and describe how they may be accessed.

BCPL (a modified version) has four sets of variables - local, static , global and external although instructions need only be provided to access the first three.

Local variables (see figure 2.1) are allocated space above the parameters on the runtime stack. They are accessed relative to P and hence we see how P serves as an index register. Notice that once the current stack frame is released, space for all local variables vanishes as well.

Static variables are accessed by reference to some label and unlike local variables remain accessible throughout program execution. The syntax of a label is simply a number followed by a colon.

External variables are accessed as if they were static variables except that information is provided for the loader so that it can resolve these references. Externals can be thought of as being another program's static variables.

Global variables are accessed relative to the G register. This space is reserved by the runtime support for the particular BCPL program and this support also initializes the G register.

2.3 Operands

The syntax of all operands is as follows.

```
[I]    [P | G | L] < integer >    |    *
```

P refers to the stack pointer. The contents of P is added to <integer> to obtain the address of a particular variable (parameter or local) on the current stack frame. G is interpreted similarly except that it refers to the global pointer. L denotes a particular label (e.g., Ln) . The operand in this case is treated as the address of the particular label. I means indirection. The operand determined thus far is dereferenced once. The * refers to the top of the stack(i.e., the location pointed at by register S). In this case the contents of S-1 becomes the operand and S is decremented by 1. Some examples follow:

P2	address of cell at offset 2 from P
IP2	value of cell at offset 2 from P
L3	address of cell denoted by label 3
IL3	value of cell pointed to by label 3 (e.g. value of a static variable)
*	value of a temporary variable

2.4 Operations

a. Variable access operators

four operations are used ; Load (LD), store (STORE), stack and load (STKLD), and select field and store (SLCTST).

```
LD          operand      ACC := operand
STORE       operand      location(operand) := ACC
STKLD       operand      !S := ACC
                         S := S + 1
                         ACC := operand
```

SLCTST	fieldselector	select appropriate field of the value in the accumulator and store them in the correct field of the cell specified by its address at the top of the stack. (i.e !(S-1)) Then decrement S by 1.

Some sample variable loads and stores are illustrated.

temporary:	LD *	
local:	LD IPn	STORE Pn
static:	LD ILn	STORE Ln

Notice that LD Pn loads the address of the local variable <u>not</u> the value.

b. Diadic expression operators

All operators can be defined as follows.

op operand	ACC := ACC op operand

Integer operators -
MULT,DIV,REM,PLUS,MINUS

MULT, DIV, PLUS and MINUS are as expected. REM is the integer remainder on the division of the ACC by the operand.

Relational operators -
EQ,NE,LS,GR,LE,GE

EQ and NE are equal and not equal. LS and GR are less than and greater than. LE and GE are less than or equal to and greater than or equal to.

Logical operators -
LSHIFT,RSHIFT,LOGAND,LOGOR,EQV,EXOR

L and RSHIFT are the left and right shift operators. LOGAND and LOGOR are the logical AND and OR. EXOR is the exclusive OR or bitwise non-equivalence. EQV is bitwise equivalence.

Bit operators -
SLCTAP

This applies to field selectors in the BCPL sense. The appropriate field is selected.

c. S register manipulation

To allow for flexible manipulation of the S register a combination of monadic and diadic operators are defined. This allows one to set the S register in a relative (i.e. SREL, SRELI) or absolute sense. (SSET, SSETI) . If one allows an extended BCPL in which dynamic storage allocation is implemented then it is mandatory that the S register be manipulated in a relative sense. It is true that all local variables are in a sense dynamically allocated but at present the sizes of vectors must be fixed at compile time. This ability to determine run time sizes of vectors is what we mean by dynamic allocation.

SGET	ACC := S
SSET	S := ACC
SREL	S := S + ACC
SSETI	S := operand
SRELI	S := S + operand

d. Monadic operators

NEG	ACC := - ACC
NOT	ACC := not ACC
DEREF	ACC := !ACC
PUSH	!S := ACC ; S := S + 1
POP	ACC := !(S - 1) ; S := S - 1
TRUE	ACC:= TRUE
FALSE	ACC := FALSE
FINISH	FINISH

e. Transfer

GOTO	C := ACC
JUMP operand	C := operand

```
JT operand    C := (ACC = TRUE) -> operand, C
JF operand    C := (ACC = FALSE) -> operand,C
```

The switchon statement is implemented by the SWITCHON command;

SWITCHON k Ld k1 l1 k2 l2 ...... kk lk

The accumulator controls the switch; it examines the k case constants in left to right order and when a match occurs then a jump is made to the corresponding case label, otherwise a jump is made to the default label Ld.

f. Function and routine calling

No difference is made between the call and return instructions for a function or a routine. When a function is called it returns its result in the ACC. Prior to a call space should be saved for the links, (savespacesize denotes this number) the i parameters pushed on the stack and the address of the routine loaded into the ACC.

The CALL i instruction has the following effect

```
temp := S - (i + savespacesize)
temp!0 := C
temp!1 := P
P := temp
C := ACC
```

The RTRN instruction is as follows

```
C := P!0
S := P
P := P!1
```

The routine called, on entry is responsible to set the S register so that it points as in Figure 1 (above the parameters and preliminary local variables).

g. Pseudo instructions

<number>:	denotes label <number>.
@name	denotes the start of the code for the routine called name.
SECTION "section name"	indicates the start of the code for the section "section name".
END	indicates the end of a section's code

h. Data reservation instructions

There is one general purpose directive used to reserve space. This is the DATA operator. Its operands can be numbers, characters (enclosed in single quotes), strings (enclosed in double quotes) or labels. It reserves space in the subsequent cells for its operands.

2.5 An Example

The following program and its SLIM code will serve as an example.

```
||
||
|| this is the sample BCPL program
||
let f(a, b, c) be
   {
   let v = vec 4
   and w = 0
   v!1 := (a + b) *(b + c)
   }

and start() be
   f(1, 2, 3)

||
|| this is the translated version of the above program into SLIM:
|| It is an exact reproduction of the slim compiler output.
||
  SSETI P2 JUMP L5
@f
1:   SSETI P5 LD P7  STKLD 0  PUSH   SSETI P12 LD IP5 PLUS 1
     STKLD IP3 PLUS IP4  STKLD IP2 PLUS IP3 MULT * STORE
 *   RTRN
@start
3:   SSETI P2 SRELI 2 LD 1  STKLD 2  STKLD 3
 STKLD IL2 CALL 3 RTRN
 5:  FINISH
2: DATA  L1
4: DATA  L3
END
```

This example serves to illustrate three features in SLIM: the call mechanism; vector allocation and expression evaluation.

i. The call mechanism

The prelude before the actual call involves saving space for the links and evaluating the parameters. Linkage space is saved by the instruction SRELI 2. Here two cells are left to contain the previous stack frame pointer and program counter. The three parameters are evaluated and the address of "f" loaded into the ACC. At this point the stack is as in Figure 2.2.

```
┌────┬─────┬─────┬──┬──┬──┬────┐
│    │ old │ new │1 │2 │3 │    │
│    │links│links│  │  │  │    │
└────┴─────┴─────┴──┴──┴──┴────┘

     ▲                  ▲
     │                  │
     │                  │
     P                  S
```

Figure 2.2 Stack prior to call

The effect of the CALL 3 instruction can be pictured as in Figure 2.3.

```
┌────┬─────┬─────┬──┬──┬──┬────┐
│    │ old │ new │1 │2 │3 │    │
│    │links│links│  │  │  │    │
└────┴─────┴─────┴──┴──┴──┴────┘

           ▲            ▲
           │            │
           │            │
           P            S
```

Figure 2.3 Stack after call

Notice how the P pointer has changed and we are now executing with a new stack frame. As mentioned previously, the routine called is responsible to ensure that S actually points where it should, hence the SSETI P5 instruction. This is necessary since one can call routines with fewer parameters than they expect and if the S register is not corrected, not only will further local variable allocation be completely incorrect (within the current procedure) , but any temporaries used will map onto existing local variables and cause havoc.

ii. Local variable allocation

This involves setting variables to their initial values and also allocating space. The routine expects its environment to be as in Figure 2.4. (the numbers on the top denote stack frame offsets)

```
                2 3 4 5 6
   ┌───┬──────┬─┬─┬─┬─┬─┬────┐
   |   |links |a|b|c|v|w|    |
   |   |      | | | | | |    |
   └───┴──────┴─┴─┴─┴─┴─┴────┘


       ^                ^
       |                |
       |                |
       P                S
```

Figure 2.4 The routine's stack environment

The SSETI instruction adjusts the S register appropriately. At this point offset 5 and 6 from the current stack frame pointer reserve space for the variables v and w except that they have not been initialized. Since v by definition will contain the address of a vector of size 5 the LD P7 STKLD 0 sequence accomplishes this. w is initialized via the PUSH instruction since the previous instruction has already loaded zero into the accumulator. All is well except that we must indicate somehow that we have used 5 more cells for the vector v. SSETI P12 adjusts S to reflect this fact.

iii. Expression evaluation

At this time we will concentrate on the sequence of slim code that evaluates

(a+b) *(b+c)

LD IP3 PLUS IP4 evaluates b+c. However at this stage we need to save this result in some temporary location. The STKLD IP2 PLUS IP3 accomplishes this (via STKLD) while at the same time evaluating a+b. At this point the accumulator contains a+b and the stack contains the following:

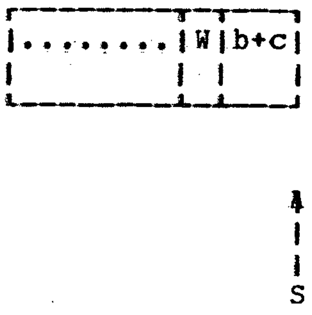

Figure 2.5
The stack in midst of expression evaluation

Now all that remains to be done is retrieve the temporary result and multiply it by the accumulator. MULT * accomplishes this and leaves the stack how it was.

Although this is not particularly convincing one must admit to the relative ease with which temporaries are handled. Chapter 3 compares the amount of code generated for the above expression using a pure stack machine with that generated by the SLIM compiler.

3. Machine Justification

In this chapter some justification for the choice of the SLIM machine is outlined. Since a normal defense would consist of responding to several questions regarding the choice of particular features, this is the form this chapter will take!

3.1 Why choose a stack machine architecture?

We will first outline a more generalized notion of what we mean by a stack machine. By a stack machine we will mean a machine in which a hardware stack plays a central role in expression evaluation, storage allocation and subroutine control and linkage. We will not require a machine to be a stack machine if and only if most instructions operate on operands held at the top of the stack.

Software has made use of stacks for a long time but most computers lack hardware stacks. As the trend to develop software in higher level languages develops we are now witnessing hardware acknowledgement of this fact with the advent of hardware stacks. The HP 3000, the Burroughs machines (B1700, B5500, B6700 and 7700) , the Data General Eclipse and the PDP-11 to a more limited extent are just some of the machines with some form of hardware stack. It is in this context of higher level language use that we will outline some of the advantages of stack machines.

A key concept in software is the subroutine. Some people still argue that effective use of subroutines (i.e., good structure) is wasteful of time and space. This is natural since as Bulman [24] says, "the subroutine call and return mechanism seems to be almost an afterthought in the architecture of many computers." The stack machine nips this argument in the bud. The best mechanism for the subroutine call and return mechanism is to involve the stack to store the return address. The stack then contains the record of the nesting of procedure calls and one no longer has to worry about saving space for the return address.

This last issue has been treated in many ways and points up another advantage of stack architectures. Often this return address has been saved in a register or worse still a local memory location. Both these methods however require extra software if one allows recursion or reentrant routines. The programmer becomes responsible for stashing this return address somewhere before the next routine (and in recursion it is the same one) is invoked. Stack architectures remove this concern from the programmer and in fact it is hard not to write reentrant programs when using a stack.

Parameters are treated efficiently in a stack architecture. What better place for them than on the stack? Many other methods that specify that space be permanently allocated to each subroutine for its parameters or that space be shared, again shift the burden for the management of this space onto the programmer. Stacking the parameters at once removes this concern from the programmer and also uses the space only when it is required.

Another key advantage of stack architectures is that they automatically provide local environments. Typically a subprogram refers to only a small subset of all identifiers declared in the whole program. In the BCPL case one only refers to local or global variables (these include statics and externals) . Since these local variables are only referenced in the procedure in which they are defined it seems wasteful to have space permanently allocated for these variables when the procedure is not active. Allocating this space on the stack in the local environment also accomplishes something else. Since the local environment is accessed relative to some environment pointer (P in SLIM's case) addresses for these variables need only specify offsets from this environment pointer. Since these offsets are typically small (95% < 10) , instructions require fewer bits. Hence program space is saved. Program space saving is also accomplished by requiring no implicit addresses for those variables that are implicit. Addresses are of two kinds in a machine: explicit- those variables explicitly mentioned by the program; and implicit - those that arise out of the need for some temporary storage location. These are automatically provided by stack architectures and their reference just involves referencing the top of the stack which requires no implicit address bits. Once again code is compacted. Global variable access also requires fewer bits since they are accessed relative to some global environment pointer.

Another advantage of stack architectures is that they exhibit the difference between program and task. Using

Organick's [26] terminology, an incarnation of a task is a combination of a time invariant algorithm(the code) and the time varying record of execution. The stack embodies this record and hence a task can be seen as code plus stack. Processes then can be easily conceived of as some code plus some stack area for the particular process. Process switching then only involves transfer of control and the provision of some space to contain the time varying record of execution.

Interrupt handling can also be treated effectively as unexpected procedure calls. Since we know the limit of the stack the interrupt can be serviced transparent to whatever was executing at the time.

3.2 Why choose a single accumulator?

Simplicity is the main reason. A single accumulator is all one really needs. Inbuilt registers like the P, S and G registers provide the index functions that one normally is provided with except that the P and G are automatically maintained. In an environment of short procedures Tanenbaum [2] concludes "the register sets provided by a third generation machine are of little value". They can be used for intermediate results but with the stack mechanism (see chapter 2) one register is sufficient. In Tanenbaum's environment where one out of every four statements is a procedure call the save-restore overhead makes it inefficient to use registers to hold local variables. When one considers what is involved in process switching the smaller the number of states associated with a process, the quicker and easier it becomes to implement process switching. Having considered why we choose to minimize the number of registers one perhaps wonders why we did not go the stack machine route completely and eliminate the ACC altogether. This will be addressed in section 3.4., but perhaps we can outline an equivalence of SLIM and an addressable top of stack location on a pure stack machine. Consider the following expression and its equivalent evaluation by three machines: SLIM, a pure stack machine and a modified stack machine as above.

```
(A + B) * (C + D)

Pure Stack          SLIM            Modified Stack


LOAD A              LD A            LOAD A
LOAD B              PLUS B          PLUS B
PLUS                STKLD C         LOAD C
LOAD C              PLUS D          PLUS D
LOAD D              MULT *          MULT
PLUS
TIMES
```

The modified stack machine can be thought of as a machine with a floating ACC in SLIM's sense. This ACC is actually the current top of stack. Comparing this and SLIM code one notices the similarity except that there are two versions of each diadic operator in the case of the modified stack machine: one that requires an operand (e.g., PLUS B & PLUS D) and one that takes both its operands from the stack (e.g., MULT) . This introduces a further complexity into the machine when one has 2 versions of each diadic operator. With 17 diadic operators this is quite significant since these extra operators have to be encoded. This might require extra bits in the opcode field for the instruction leaving less space to encode the operands. SLIM however only has one extra operator (from a stack machine's viewpoint) - STKLD. Another factor that favours the single ACC machine is the necessity of handling environments that return values. The two particular instances of this in BCPL are the function and the valof block. In both cases some result computed at the top of the current stack frame(in a pure stack or modified stack machine) must be passed to the preceeding environment while at the same time collapsing the present environment. In the function case this requires an extra operator FNRN to do precisely this. The valof block uses the RSTACK operator. This unecessarily adds to the complexity of the machine.SLIM only needs to return any value in the accumulator and hence requires no extra operators.

3.3 Why have an S register?

This register always points to the top of the stack and hence indicates the next possible unused stack location. There are three main reasons why this register is made explicit.

Interrupts can be cleanly handled since the S register always indicates where a new stack frame could begin. Hence the interrupt hardware need only fill in the links as in a normal call starting at where the S register points. The K register for the PICA-B machine [25] functions in much the same way.

Dynamic storage allocation is another major reason why one needs the S register. It is when one does not know the size of the current stack frame (e.g., with dynamic vectors) that one needs to be able to manipulate the S register in a relative manner. One cannot just use offsets from P since these offsets are only known at run time. The CALL instruction is a relative type of instruction in the sense that the "n" specifies the number of parameters passed as opposed to the corresponding INTCODE instruction K d where the d specifies the size of the callers stack frame. Standard BCPL does not allow for dynamic storage allocation (neither does the SLIM version of the compiler) but for the ease with which this could be achieved we present a BCPL fragment and the corresponding SLIM code. From this, one will hopefully appreciate the usefulness of the S register.

Extended BCPL.

```
   let v1 = vec <expr1>
   and v2 = vec <expr2>
   and c = 0
   .
   .
```

SLIM code.

```
   SSETI Pn        set S to point above space for vars
   SGET            get this value in the accumulator
   STORE Pv1       make v1 point to its space
   .
   code to
   evaluate
   <expr1>
   .
   SREL            adjust S by the value of <expr1>
   SGET
   STORE Pv2       make v2 point to its space
   .
   code to
   evaluate
   <expr2>
   .
   SREL            make S point to free space
   LD 0
   STORE Pc        initialize c
```

Notice that this code sequence differs from the example in chapter 2 since there we knew sizes explicitly at compile time and hence could compile more efficient code.

A third reason is that the S register is the means of generating and retrieving implicit variables that are required. The SLIM operators PUSH & STKLD and the operand * are the means of realizing this very valuable feature.

3.4 Why single address?

When one's prime objective in the design of a simple machine is that programs be represented efficiently, considerable thought must be given to the number of addresses an instruction should contain. The greater the number of addresses the larger the instruction size and hence a larger total program size is more likely. The number of addresses per instruction can vary from three to none in a pure stack machine. As Ibbett et el. [27] state there are a number of conflicting virtues related to the various possibilities. "Simple operations such as the setting and incrementing of variables are more concisely described by two and three address schemes. Evaluations of longer expressions are more concisely defined by zero address and one address systems, however, because the address in which the result is accumulating is implied."

By considering some sample expressions a choice was made to utilize the one address scheme. This is made possible by the provision of the STKLD instruction which first stacks the accumulator contents, together with the * operand which provides a way to access stacked partial results. This maintains the valuable features of a stack machine while providing more compact code. In the following two examples two measures are used: the number of words in the machine independent sense where there is one instruction per word; the number of bytes in the more applied sense. A comparison of the total sizes demonstrates the superiority of the one address scheme.

EXAMPLE 1. Comparison of SLIM and a stack machine

(A + B) *(C + D)

Stack machine	words	bytes	SLIM	words	bytes
LOAD A	1	2	LD A	1	2
LOAD B	1	2	PLUS B	1	2
PLUS	1	1	STKLD C	1	2
LD C	1	2	PLUS D	1	2
LOAD D	1	2	MULT *	1	1
PLUS	1	1	Total:	5	9
MULT	1	1			
Total:	7	11			

EXAMPLE 2. Comparison of SLIM and a stack machine

A + B

Stack machine	words	bytes	SLIM	words	bytes
LOAD A	1	2	LD A	1	2
LOAD B	1	2	PLUS B	1	2
PLUS	1	1	TOTAL:	2	4
TOTAL:	3	5			

Having illustrated a simple comparison above the rest of this thesis will attempt to compare the SLIM machine with other existing architectures. We will first discuss the issue of what measure to use and then demonstrate that the first objective in the design of SLIM has been achieved.

4. Measures and results

4.1 Ideal Program representation

We are now at the stage where you may be asking - so what? The context of the design has been sketched and the machine described and verbally justified. But how can we evaluate this architecture? This is what concerns us in this chapter. We will examine some general aspects of measures, describe three specifically and then proceed to use the chosen measure to compare our architecture with two other architectures.

What should be included in a measure? One obvious component is that the measure be objective; something that can be precisely quantified. Unfortunately non-quantitative measures generally tend to receive little merit. Somehow one feels that the measure should also incorporate the space time product. Space generally meaning program size, and time being some measure of hardware efficiency. However this space component could justifiably include items such as compiler size, size of the runtime support etc. Somewhere one has to draw the limit. A more imporatnt issue is concerned with whether one can evaluate architectures just on the basis of their design without any regard for what use will be made of them. Or more precisely: Can architectures be evaluated in a program independent fashion?

In the next section we will consider two not strictly program independent measures and one strictly program dependent measure.

4.2 Measures

i. Flynn's measures

Flynn [28] compares an architecture against what he takes to be an ultimately simple, fully explicit architecture. As he states: "In a simple architecture nothing is implied - no registers or counters are invisible to the problem state programmer. Each instruction contains an operation, the full generalized address specification (allowing if necessary multiple levels of indirection through tables etc.) for both source operands, a result operand, and a test of the result which selects an address for the next instruction".

He then classifies instructions into three broad categories.

M instructions are memory partition movement instructions such as the LOAD and STORE instructions which move data items within a storage hierarchy.

P instructions are procedural instructions which perform functions associated with instruction sequencing, i.e., TEST, BRANCH, COMPARE etc., but perform no transformation on data.

F instructions perform computational functions in that they operate on data. They include arithmetic operations of all types, as well as logical and shifting operations

To Flynn M and P instructions are overhead instructions whereas F type instructions are the only ones that do any work. Therefore the three ratios to measure this overhead are:

1. M ratio: ration of M to F instructions
2. P ratio: ratio of P to F instructions
3. NF ratio: ratio of the sum of M and P instructions to F instructions

An ideal machine would have M = P = NF = 0. Flynn uses these ratios to evaluate the IBM 7090, system 360 and the PDP 10.

ii. Instruction mixes

This is the frequency distribution of the types of instructions executed during the processing of a workload. The best known published example is the Gibson mix. Gibson obtained frequencies in this mix from an analysis of the use of instructions in technical and scientific applications in IBM 7090 installations. Flynn has obtained a mix appropriate to system 360 installations. These mixes are used to evaluate architectures primarily by providing time measures. The frequency of instruction use in the particular class is multiplied by the average instruction execution time in this class and these summed for all classes in the mix. The result of average instruction execution time is taken to be a measure of the architecture and used for comparison purposes.

iii. Program representation size

Given a program or a representative set of programs in some high level language, one translates these programs to machine language programs for various machines. The space required by the object programs is used for comparison among the various machines. The smaller the space required for the code the better the machine architecture according to this measure. This measure is used by Tanenbaum and is the one we will use and justify shortly.

At this point one needs to recognize that in some high level language translations the machine code contains a large number of implicit as opposed to explicit subroutine calls to built in library functions. Explicit calls to library functions are those which the program directly specifies. As a result the machine code may be small but the percentage of implicit subroutine calls may be high. What this actually points out is that the code for the built in library functions is the microcode for the instructions required by the higher level language. This reflects the fact that the machine at the current level is not suited to the particular language. For this same measure to be used in cases like this, each implicit library call should count for the number of words in the code of that library call, not just as one subroutine call.

We will now briefly comment on these measures in light of the question raised previously: Can architectures be evaluated in a program independent fashion? The underlying issue here is to guage how effectively the machine accomplishes its purpose. By machine we mean a configuration of the micro architecture that realizes an instruction set. In many cases this configuration is hardwired but in others (e.g., the B1700) one can dynamically reconfigure the micro architecture. By purpose we mean how the machine facilitates what people want to do. This of course is accomplished at a number of levels: modes of expression available (i.e., programming languages); software packages; operating systems; programming environments (batch or timeshared) etc. What we are more concerned with here, is the primary level concerning modes of expression. What people want to do is most often expressed algorithmically in some high level language. Thus programs written in a high level language are the primary vehicle of conveying people's intent to the machine. Therefore we will assume that programs in some programming language or languages are a good indication of the use of a machine. Note that we are not tying the expression of algorithms to one particular language. Rather we are suggesting that much has been learned about algorithms and ways to represent them in programming languages. This makes programs in a given class of languages representative of what people want to do, and hence machines should be evaluated with respect to a given class of languages. From this perspective the use of the machine is the common denominator in an evaluation <u>not</u> some general notions of machine design. We are now left with the question of how to effectively and precisely measure how well mapped the machine is. By well mapped match we mean how concisely transformations (or state transitions) in the high level language are represented in the lower level machine. The more concise this representation the better mapped the machine. This is the bias we have in choosing our measure of program representation size (i.e, code space).

Flynn's measures clearly emphasize the functional characteristics of an architecture. He attempts to compare archtectures solely from the functional architectural viewpoint. He is not stricty comparing architectures in a program dependent manner. One however could possibly argue that his simple machine is actually the most optimal representation for programs provided one accepts his definition of optimality. (i.e. no overhead) This measure however is more concerned with validating or invalidating the following thesis: Machine design has strived towards decreasing memory references, (e.g., of instructions and their operands) but this has introduced considerable overhead. This overhead is a result of making several explicit functions (in Flynn's simple machine case) implicit. Two cases of this overhead are:

i. The treatment of programs as linear strings and consequently maintaining the program counter implicitly.

ii. The introduction of registers to hold operands in local store and not in main memory.

The former case has introduced the whole range of branch instructions whereas the latter has introduced the Store and Load variations. After making some measurements of various computer architectures Flynn concludes that in fact the overhead is considerable. As we can see, the emphasis in Flynn's measures of measuring this overhead is not directly concerned with how well mapped the machine is. Therefore we will not use it.

Instruction mixes basically are a test of hardware. After one has derived a suitable mix one is generally interested in average instruction execution time or some such time oriented measure. Although this is useful it subtly incorporates the variables of the technology used and the encoding of instructions. This latter variable greatly affects the complexity of the microcode and hence the speed. (We assume a microcoded implementation of an instruction set.) These variables do not really give an indication of how well mapped the machine is.

Even a fast average instruction execution time does not necessarily guarantee anything. If all one has is fast instructions that do nothing, the increase in instructions needed to dc something useful will definitely detract from any advantage speed might have initially provided. In other words the power of an instruction is not necessarily taken into account. This power is representative in some sense of what you wculd like to do and since instruction mixes measure this poorly we will not use this measure either.

Also because different machines (and hence instruction sets) produce different user characteristics, it is not clear that the same instruction mix is applicable to all machines under consideration.

We will now outline the reasons for our choice of the size of programs as our measure. Small representation of programs (i.e., code space) clearly reflects a well mapped machine. If some other machine requires more code space for the same program then this is indicative of the need for more state transitions in the machine than actually required by the program. In other words the machine is partially mapped. In our specific case we wish to show that by this measure for the language BCPL, the SLIM machine is a well mapped machine, well suited to BCPL.

Secondly size and space are intertwined. The smaller the program the faster the interpretation is likely to be. Small representation of programs also has a third more practical advantage. In this age of mini and micro computers the ability to run large programs is a great advantage with the limited memory these systems usually provide. Fourthly a decrease in program size can lead to an increase in the degree of multiprogramming and potentially decrease the page fault rate.

It is for this combination of architectural and practical considerations that we use space as the measure for comparison in the following sections.

4.3 Results

Now having established our measure for the purpose of comparison we will proceed to compare the SLIM machine against two architectures. These are OCODE (see [21]) and EM-1 (see [2]) . These machines will not be described but one is referred to their adequate description elsewhere.

i. OCODE versus SLIM - round 1

OCODE is the classical first step in the translation of BCPL programs. From the Applicative Expression Tree (AE tree) representation of the BCPL program OCODE is generated. OCODE is a stack machine and this is one of the reasons why we have chosen it as one of the machines for comparison. In some sense it is representative of stack machines for which there is wide respect. The second reason for choosing OCODE is that it was especially designed for the translation of BCPL.

The procedure for comparison has involved translating approximately 8500 lines of BCPL source into OCODE and SLIM code. In fact BCPL is not translated into OCODE but into BCODE. The only difference between the two is that BCODE is intended to be used as a real machine and so OCODE instructions are encoded and object modules generated. BCODE is the work of a local, unpublished project at the University of British Columbia.

One might object here that encoding has not been mentioned. At this level of comparison, instructions that take one word in BCODE occupy the same in SLIM. Double word instructions will occur more frequently for SLIM since there is less space for encoding operands. Therefore for the measure of code sizes encoding can be treated as a constant in this case and not enter into the comparisons.

Two measures are used: number of instructions and code sizes. The following table describes the programs used and gives the ratios of BCODE to SLIM for both measures.

PROGRAM	INSTRUCTIONS	CODE SIZE
Intcode interpreter	1.18	1.20
(2 sections)	1.22	1.17
Intcode assembler	1.09	1.13
(3 sections)	1.18	1.15
	1.12	1.10
BCPL compiler	1.14	1.10
(6 sections)	1.11	1.12
	1.12	1.10
	1.10	1.10
	1.05	1.15
	1.14	1.20
OCODE to 370 assembler	1.15	1.15
(5 sections)	1.15	1.11
	1.14	1.11
	1.18	1.12
	1.13	1.11
Text editor	1.03	1.06
(4 sections)	1.06	1.06
	1.05	1.05
	1.06	1.04
Average:	1.12	1.12

TABLE I. OCODE and SLIM comparison

As can be seen there is a twelve percent gain on the average for the SLIM machine using this measure.

ii. EM-1 versus SLIM - round 2

This machine is a recent attempt to provide a machine that will provide very compact representation for a large class of languages. For example ALGOL 60, ALGOL - 68 , Pascal, XPL, BCPL, SAL etc.

In Tanenbaum's paper [2] he compares four programs and their code sizes on the EM-1, PDP-11 and Cyber. He gets ratios as low as 1.5 with the PDP-11 and as high as 6.3 on the Cyber.

Thus this machine is well suited as a comparison with SLIM. In order to perform the comparison we must first provide a more compact encoding for SLIM to match EM-1's encoding. The encoding is presented in Appendix I. Appendix II contains the BCPL source for three programs used for the comparison. Since there is no BCPL to EM-1 compiler, equivalent C programs were provided and these too are included. The following table presents the results.

	OPTIMIZED EM-1 (bytes)	SLIM (bytes)
Hanoi	46	41
Bubblesort	80	81
Expression	20	27

TABLE II. Optimized EM-1 and SLIM comparison

The three programs were chosen to represent 3 classes of program : procedure calling (towers of hanoi) ; general loop mechanisms (bubblesort) and expression evaluation . Before one concludes too much here, where SLIM does not outperform EM-1 dramatically we should be aware of a number of characteristics of "optimized" EM-1 code. It is very closely tied to language directed machine design. Two examples of this optimization fcllow:

i) Due to a fair amount of incrementing by 1 in higher level languages EM-1 provides an increment operator. Since SLIM does not, the equivalent SLIM code (LD IPn PLUS 1 STORE Pn) occupies 4 bytes as opposed to 1. If this operator were provided then the 3 bytes we would save in SLIM code for the Bubblesort routine would make SLIM code more compact than EM-1 code in this case.

ii) Optimized EM-1 code recognizes consecutive lcads and replaces them by a single LOAD DOUBLE instruction. For example, instead of generating LOAD A LOAD B it generates a LOAD DOUBLE A. In our expression program there are 5 cases where this occurs: (a+b) , (c+d) , c+d, (a+b) and a+b. If this procedure had been written with the order in these expressions reversed then the results would have been significantly different. One need only note that 24 bytes are required for a pure stack machine for the expression evaluation alone and this does not take into account the procedure entry and exit.

The following table presents the results assuming the lack of the above two optimizations for EM-1 and also that procedure entry and exit occupy three bytes as in SLIM.

	EM-1 (bytes)	SLIM (bytes)
Hanoi	46	41
Bubblesort	83	81
Expression	27	27

TABLE III. EM-1 and SLIM comparison

Despite the lack of any EM-1 type optimization in SLIM the machines compare very favourably. We now present our observations and directions for further research.

5. Conclusions and directions for further research

Although this thesis has presented the design of an intermediate machine suited to a particular high level language, not much has been said about the various approaches to instruction set design. We will now outline some approaches to instruction set design and then make some comments on the particular approach used in this thesis.

Lipovski and Doty [30] describe three schools of thought on instruction set design. The oldest approach is to use statistics based on coding experience with an older architecture, to assist in constructing a more refined machine. Instructions that are frequently used are made faster and perhaps more flexible. Statistically significant instruction sequences are made into primitive operations. The second approach is to choose a widely used high level language. The primitive operations necessary to execute this high level language are identified, and then realized in the instruction set. The third approach identifies a range of problems to be solved using the computer and a set of characteristics of the technology to be used to realize the machine. The problems to be solved are treated as 'axioms', (premises) and the decisions leading up to the design of the architecture are treated as 'theorems' (implications). The 'proof' gives all the reasons for the specific design decision (implication) in terms of the problems to be solved (premises) and earlier implications. Clearly the approach used in the design of SLIM is the high level language approach. These approaches all have their pros and cons.

The statistical approach generally assures some form of compatibility between between the old and the more refined machine. This is convenient corporate policy but can tend to entrench existing patterns of operation and insight and not allow for new innovations. The high level language approach is more suited to the more common forms of expression but is generally applied to one specific high level language. Since most computers run more than one language, what is optimal for one language may not be optimal for another. There are two ways to overcome this problem. One is to allow for various microcoded intermediate languages as in the Burroughs B1700. The other is to design instruction sets that are well suited to a number of languages. The EM-1 machine is one signpost in this direction. The premise-implication approach requires careful thought for all design decisions and hence makes it difficult to write the description. However this approach perhaps shows more clearly what the system is intended for and what its limitations are.

We will now make some conclusions regarding the methodology used in the design of SLIM and the results obtained. The results clearly show that the objectives governing the design of SLIM have been achieved. Using the measure of program representation size SLIM compares very favourably with a number of architectures. SLIM is a definite improvement over OCODE and is approximately equivalent to the EM-1 machine. Although no mention has been made of INTCODE, one automatically can infer from the SLIM versus OCODE results that the SLIM representation of programs is much smaller than their INTCODE counterparts. The objective of simplicity in machine architecture also has been realized. The achievement of these objectives show that useful work can be done within this particular approach to instruction set design. Regarding the approach itself it is difficult to be specific. Although we have argued elsewhere for the importance of this approach it is difficult to provide handles to assist in synthesizing the operations necessary to execute high level languages. One not only has to determine operations but one must first of all determine the architectural building blocks on which these operations will operate. There are a number of accepted building blocks in existence. For example the importance of stacks in environment allocation, procedure calling and expression evaluation. This is one area of further research where similar work with other languages might distill other architectural building blocks. This in turn will help to identify the primitive operations necessary to execute high level languages.

Another approach we have not mentioned that differs from that of instruction set design is direct execution of high level languages. In this approach the machine instruction set becomes the operations of the high level language. This approach also has a number of pros and cons. It eliminates the compilation process, speeds up execution of programs and generally provides greater program density. On the other hand the size of the microprogram to interpret the high level language instructions will be large and very complex . With current technology and costs the construction of such a machine would be prohibitively expensive. The machine also by definition will be very special purpose. Since users may want to use other languages he may find it awkward to compile them into the base high level language. The representation of these other high level language programs in the base language may also be large and their execution slow. More importantly, this approach depends on how suited the language is to interpretive execution. In this mode of execution each statement is decoded just before it is used. BCPL in its pure source form is definitely not suited to this approach. For example a procedure call that involves a procedure that is defined 3000 lines further on in the source, cannot be immediately executed. For BCPL to be executed in this manner some form of intermediate program representation would be necessary. This borders closely on the approach we have used. Two areas of research arise out of considering this approach as it applies to languages like BCPL. One is to develop suitable high level intermediate representations that can be directly executed. The other is to develop language design principles that will provide languages that can be directly executed.

The final issue that concerns us is the development of suitable measures for architecture comparisons. The choice of methodology in instruction set design clearly biases the choice of measure. For example, those adopting the statistical approach might be more interested in time oriented measures. However we have argued earlier for the importance of the high level language approach to instruction set design and therefore conclude that our measure of program representation size is an important component of any measure that is devised. Of course our measure has a number of deficiencies. It is dependent on the efficiency of the translation section of the compiler used. Comparisons are meaningful only if the translation sections of the various compilers use the same optimizations. This is sometimes difficult to achieve. Program representation size is also just one component of a measure. Though this measure has been useful for our comparison purposes, this subject of measures for evaluation purposes requires further work and study to produce a more comprehensive measure.

1. Abd-alla, A.M & Karlgaard, D.C. Heuristic synthesis of microprogrammed computer architectures. IEEE transactions on Computers, Vol C-29, No. 8, Aug 1974.

2. Tanenbaum. A.S. Implications of structured programming for machine architecture. CACM, Vol 21, number 3, March 1978 pp237-246

3. Knuth, D.E. An empirical study of FORTRAN programs. Software practice and experience 1 (1971), pp261-301

4. Alexander, W.G. How a programming language is used. CSRG-10 , U. of Toronto, Ontario, Canada

5. Wortman, D.B. A study of language directed computer design. CSRG-20, U. of Toronto, Toronto, Ontario(1972), Canada.

6. Salvadori, A., Gordon, J.& Capstick, C. Static profile of COBOL programs. Sigplan notices (ACM) 10(1975), pp20 - 33

7. Wichmann, B.A. ALGOL 60 Compilation and assessment. Academic Press, London and New York, 1975

8. Agrawala, A.K. & Rauscher, T.G. Foundations of microprogramming: Archtecture, Software and Applications. Academic Press. 1976

9. Salisbury, A.B. Microprogrammed Computer Archtectures. New York : Elsevier. 1976

10. Marshland, T.A. & Demco, J.C. A contemporary computer emulation. Technical report TR76-1, Feb. 1976, Dept. of Cpsc., University of Alberta, Canada.

11. Wilner, W.T. The design of the Burroughs B1700. Proc. of the AFIPS FJCC, Vol. 41, AFIPS press, Montvale N.J., 1972, pp 489 - 497

12. Saal, H.J. On measuring computer systems by microprogramming in [29]

13. Rosin, R.F. Systems Architecture and Microprogramming: some comments in [29]

14. Appelbe, W.F. A semantic representation for translation of high-level Algorithmic Languages. PhD Thesis, U. of British Columbia, Vancouver, Canada, 1978

15. McWilliams, T.M. & Fuller S.H. & Sherwood, W.H. Using LSI processor bit slices to build a PDP-11 - a case study in microcomputer design . Proc. AFIPS NCC 1977. pp 243-253.

16. Draft as of Nov. 15, 1977 of the Objectives for computer

science. Local memo from the Dept. head D.C. Gilmore to faculty and graduate students

17. Peck, J.E.L. The essence of computer science, UBC, Vancouver, 1975

18. Peck, J.E.L., Manis, V.S. & Webb, W.E. Code compaction for minicomputers with INTCODE and MINICODE. Technical Report 75-02, Dept. of Computer Science, U. of British Columbia, Vancouver, Canada

19.Stoy, J.E. & Strachey, C. OS-6 - an experimental operating system for a small computer. The computer Journal, 15, Nos 2 & 3, 1972.

20. Richards, M. BCPL: a tool for Compile writing and System programming. Proc. of the AFIPS 1969 SJCC Vol 34, AFIPS press, Montvale, New Jersey (1969), pp 557-566

21. Richards, M. The portability of the BCPL compiler. Software Practice and Experience, 1:1(1971), pp135 - 146

22. Richards, M. Bootstrapping the BCPL compiler. in Van der Poel, W.I, & Maarsen, I. (eds.) Machine oriented higher level languages. North Holland and American Elsevier,1974

23. Ritchie, D.M. & Thompson, K. The UNIX timesharing System. CACM , Vol. 17 No. 7, (july 1974), pp365-375

24.Bulman, D.M. Stack Computers:An introduction. Computer, May 1977. pp 18-28.

25. Abramson, H., Fox,M., Gorlick,M., Manis,V. & Peck, J. The PICA-B computer. An abstract target machine for a transportable Single-User Operating Environment. submitted for publication.

26.Organick, E.I. Computer system organization, The B5700/B6700 Series. ACM Monograph Series, Academic Press (1973)

27. Ibett, R.N. & Capon, P.C. The development of the MU5 Computer system CACM vol 21 no 1 Jan 1978. pp 13-24.

28.Flynn, M.J. Computer organization and architecture. Lecture notes for the Advanced course on operating systems Munich, Germany. July 28 to August 5, 1977.

29. Boon, C. (ed.) Microprogramming and System architecture. INFOTECH state of the art report 23, Maidenhead, Berkshire, U.K., 1975.

30. Lipovski, G.J. & Doty, K.L. Developments and Directions in Computer Architecture. Computer, Aug. 1978. pp 54-67.

Towards a single byte encoding

This appendix contains the encoding breakdown for SLIM which fits the opcode in cne byte and the operand (if any) in the following byte. Double word instructions would have 255 in the first byte which would singnify that the following three bytes contain the instruction - 1 for the opcode and 2 for the operands since this is a double word instruction.

We will first examine the number of operands required for the various operators and outline the distribution of opcodes. Since we only have a one byte opcode many operators may have nine encodings to account for the nine possible operands.

OPERAND TYPE	NUMBER OF VARIANTS	(SYMBOLIC FORM)
global	2	IG, G
local	2	IP, P
static	2	IL, L
top of stack	1	*
relative address	2	IR, R
TOTAL:	9	

OPERATORS THAT COULD TAKE ALL NINE VARIATIONS
mult, div, plus, minus,
eq, ne, ls, gr,
le, ge, lshift, rshift,
logand, logor, exor, ld,
stkld, store, rem, eqv
SUB TOTAL: 20x9 = 180

OPERATORS THAT DO NOT TAKE ALL NINE VARIATIONS

sseti	- absolute and stack relative (2)	- 3
sreli	- absolute and stack relative (2)	- 3
call	- absolute	- 1
jump	- relative(2) , static(2)	- 4
jt	- "	- 4
jf	- "	- 4
switchon	- absolute	- 1
slctap, slctst		- 2

SUB-TOTAL: 22

OPERATORS THAT ONLY TAKE ONE VARIATION
goto, neg, not, deref, push, pop,
sset, sget, srel, finish, rtrn,
true, false
SUB-TOTAL: 13

SPECIAL ENCODING

LD IPn	1<= n <= 10	10
STKLD IPn	"	10
STORE Pn	"	10
CALL n	0<= n <= 5	6

TOTAL: 180+22+13+36 = 251

Three equivalent BCPL and C programs

```
|| Check procedure calling mechanism. The classic towers of hanoi.

global { Writef:50 }
let Hanoi( n, s, i, d) be
   {
   if n = 0 then return
   Hanoi( n-1, s, d, i)
   Writef("Move %N from %C to %C*N", n, s, d)
   Hanoi( n-1, i, s, d)
   }


|| Bubblesort. General test of loop mechanisms

manifest { falsevalue = 0 ; truevalue = 1 }
let Bubblesort(a, n) be
   {
   let sorted = falsevalue
   and LastValue = n
   and temp = 0
      {
      LastValue := LastValue - 1
      sorted := truevalue
      for j = 0 to LastValue do
         if a!j < a!(j+1)
         then
            {
            temp := a!j
            a!j := a!(j+1)
            a!(j+1) := temp
            sorted := falsevalue
            }
      } repeatwhile ( sorted = falsevalue) | ( LastValue ¬= 1 )
   }


|| Expression evaluation.

let StupidProgram( a, b, c, d) be
   {
   a := (a+b)*(c+d)
   b := c+d
   c := (a+b)/d
   d := a+b+c
   }
```

|| and now for the C version of each of these three programs

```
/* towers of hanoi */

hanoi(n, s, i, d)
char s, i, d ;
{
   if ( n == 0 ) return ;
   hanoi( n-1, s, d, i) ;
   printf("move %d from %c to %c n", n, s, d) ;
   hanoi(n-1, i, s, d) ;
}

#
#define false 0
#define true 1
/* simple bubblesort routine */

bubblesort( a, n)
int a[] ;
{
   int sorted, lastvalue, temp, j ;

   sorted = false ;
   lastvalue = n ;
   do {
      lastvalue = lastvalue -1 ;
      sorted = true ;
      for ( j = 0 ; j <= lastvalue ; j = j + 1 )
         if ( a[j] < a[j+1] )   {
            temp = a[j] ;
            a[j] = a[j+1] ;
            a[j+1] = temp ;
            sorted = false ;
         }

      } while ( ( sorted == false ) || ( lastvalue ¬= 1 ) ) ;
}

/* a stupid program that evaluates expressions */

stupidprogram( a, b, c, d)
{
   a = (a+b)*(c+d) ;
   b = c + d ;
   c = (a+b)/d ;
   d = a+b+c ;
}
```

SLIM system software

This appendix contains a brief description of the SLIM system software. This includes :

i. a BCPL to SLIM compiler

ii. a SLIM assembler

iii. a SLIM loader and interpreter

This allows one to compile and run BCPL programs. We will describe this software briefly and then illustrate the whole system on the eternal towers of hanoi!

The compiler is as expected. It allows some Vancouver extensions (e.g. for operators like '%', '+:=' etc.) . The assembler generates load modules and also performs compaction making jumps and references relative if possible. This usually saves from 5 to 10 percent of the program size. The technique is the same as that described by Peck et al. [18]. All the above software is written in BCPL so that protability is enhanced.

We now present the eternal TOWERS OF HANOI right from the BCPL source to SLIM interpretation. This is an edited version of a live MTS session at UBC.

```
# COMMENT LIST OF THE SOURCE
# LIST -HANOI
>       1      SECTION. "HANOI"
>       4      GET. "FOX:BCPLHDR"
>       4.5    ENTRY $( START:"START" $)
>       5      LET HANOI(N, S, I, D) BE
```

```
 >     6       $( IF N <= 0 THEN RETURN
 >     7          HANOI(N-1, S, D, I)
 >     8          WRITEF("MOVE %N FROM %C TO %C*N", N, S, D)
 >     9          HANOI(N-1, I, S, D) $)
 >    10
 >    11      AND START() BE
 >    12      $( LET N = 0
 >    13         WRITES("ENTER NUMBER*N")
 >    14         N := READN()
 >    15         WRITEF("NUMBER INPUT WAS %N*N", N)
 >    16         IF N <= 0 THEN FINISH
 >    17         HANOI(N, 'S', 'I', 'D')
 >    18         $) REPEAT
 # END OF FILE
 # COMMENT COMPILE IT
 # RUN BCPL.COMPILER T=1S SCARDS=-HANOI PAR=I
 # EXECUTION BEGINS
   BCPL/SLIM (1978 MAY)
   PARAMETER = 'I'
   LOGICAL UNIT '0' WAS NOT SPECIFIED; -OC# ASSUMED.
   LOGICAL UNIT '10' WAS NOT SPECIFIED; -STATS ASSUMED.


   SECTION HANOI


   COMPILATION COMPLETE; 0 ERRORS DETECTED
 # EXECUTION TERMINATED
 # COMMENT LIST THE SLIM CODE
 # LIST -OC#
    SECTION HANOI

    EXTERNAL L1 "WRCH"
    EXTERNAL L2 "RDCH"
    EXTERNAL L3 "WRITEO"
    EXTERNAL L4 "WRITED"
    EXTERNAL L5 "WRITEHEX"
    EXTERNAL L6 "WRITEOCT"
    EXTERNAL L7 "WRITES"
    EXTERNAL L8 "WRITEF"
    EXTERNAL L9 "READN"
    EXTERNAL L10 "WRITEX"
    EXTERNAL L11 "NEWPAGE"
    EXTERNAL L12 "NEWLINE"
    EXTERNAL L13 "WRITEN"
    EXTERNAL L14 "PACKSTRING"
    EXTERNAL L15 "UNPACKSTRING" SSETI P2 JUMP L20
    @HANOI
    17:  SSETI P6 LD IP2 LE 0  JF L21 RTRN
    21:  SRELI 2 LD IP2 MINUS 1   STKLD IP3  STKLD IP5
    STKLD IP4  STKLD IL18 CALL 4 SRELI 2
    LD L22  STKLD IP2  STKLD IP3  STKLD IP5  STKLD IL8
    CALL 4 SRELI 2 LD IP2 MINUS 1
    STKLD IP4  STKLD IP3  STKLD IP5  STKLD IL18
    CALL 4 RTRN
    22: DATA "MOVE %N FROM %C TO %C*N"
    @START
    19:  SSETI P2
    23:  LD 0  PUSH  SRELI 2 LD L24  STKLD IL7 CALL 1
    SRELI 2 LD IL9 CALL 0
    STORE P2 SRELI 2 LD L25  STKLD IP2  STKLD IL8 CALL 2
    LD IP2 LE 0  JF L26
    FINISH
    26:  SRELI 2 LD IP2  STKLD 'S'  STKLD 'I'  STKLD 'D'
    STKLD IL18 CALL 4  SSETI P2 JUMP L23
    RTRN
    25: DATA "NUMBER INPUT WAS %N*N"
    24: DATA "ENTER NUMBER*N"
    20:  FINISH
    16: DATA  L19
    18: DATA  L17
    ENTRY L16 "START"
    END
 # END OF FILE
 # COMMENT ASSEMBLE IT
 # RUN ASM T=1S SCARDS=-OC#
 # EXECUTION BEGINS

   PARAMETER ''

   SPUNCH DEFAULTS TO '-CODE#'

   S L I M ASSEMBLER ( VERSION 3. JULY 1978 )
 # EXECUTION TERMINATED
 # COMMENT LIST THE LOAD MODULE
 # LIST -CODE#
    ENTRY "START" 000146
  111002  126400 +000145  111006  077002  040000  135402  174017
  114002  077002  014001  103003  103005  103004  102410 +000147
  120004  114002  075421  103002  103003  103005  102410 +000000
  120004  114002  077002  014001  103004  103003  103005  102410
 +000147  120004  174017  013324  153345  142500  066325  040306
  154726  152100  066303  040343  153100  066303  012400  111002
  074000  174005  114002  075453  102410 +000000  120001  114002
  076410 +000000  120000  105002  114002  075426  103002  102410
 +000027  120002  077002  040000  135402  174016  114002  077002
  102400  000342  102400  000311  102400  000304  103431  120004
  111002  125537  174017  012325  162324  141305  154500  144725
  153744  161500  163301  161100  066325  012400  006705  152743
  142731  040325  162324  141305  154425  174016 +000057 +000003
    EXTERNAL "WRITES" 000065
    EXTERNAL "WRITEF" 000100
    EXTERNAL "READN" 000071
    END
 # END OF FILE
 # COMMENT NOW RUN THE LOADER/INTERPRETER WITH THE LIBRARY
 # RUN INT T=1S SCARDS=-CODE#+BCPLLIB
 # EXECUTION BEGINS

   - S L I M - INTERPRETER/LOADER. VERSION 3 ( JULY 1978 )

   650 WORDS LOADED

   LOAD MAP

   000146 : "START"
   001172 : "WRITES"
   001175 : "WRITEF"
   001202 : "READN"
   001173 : "UNPACKSTRING"
   001174 : "PACKSTRING"
   001176 : "WRITED"
   001177 : "WRITEN"
   001200 : "NEWLINE"
   001201 : "NEWPAGE"
   001203 : "WRITEOCT"
   001204 : "WRITEHEX"
   001205 : "WRITEO"
   001206 : "WRITEX"
   001207 : "RDCH"
   001210 : "WRCH"
   001211 : "TERMINATOR"

   EXECUTION BEGINS
   ENTER NUMBER
   3
   NUMBER INPUT WAS 3
   MOVE 1 FROM S TO D
   MOVE 2 FROM S TO I
   MOVE 1 FROM D TO I
   MOVE 3 FROM S TO D
   MOVE 1 FROM I TO S
   MOVE 2 FROM I TO D
   MOVE 1 FROM S TO D
   ENTER NUMBER
   2
   NUMBER INPUT WAS 2
   MOVE 1 FROM S TO I
   MOVE 2 FROM S TO D
   MOVE 1 FROM I TO D
   ENTER NUMBER
   -1
   NUMBER INPUT WAS -1

   EXECUTION TERMINATED. ( 12892 INSTRUCTIONS )
 # EXECUTION TERMINATED
```