THE PORTABILITY OF BCPL TO THE MC68000
VIA THE INTERMEDIATE LANGUAGE SLIM

by

ERIC ALFREY THIRD RANGER

B.Sc., The University of British Columbia, 1980

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in
THE FACULTY OF GRADUATE STUDIES
THE DEPARTMENT OF COMPUTER SCIENCE

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA
April 1982

© Eric A. Ranger III, 1982
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date 27 April 1982
Abstract

This thesis describes the design of a system that converts the intermediate language SLIM, into assembler code for the Motorola MC68000. This is then assembled and loaded onto a MC68000 simulator where it successfully runs. The suitability of SLIM as an intermediate language is also studied, with a treatment on code optimization.
Table of Contents

Abstract .......................................................... ii
Table of Contents .................................................. iii
List of Figures ...................................................... v
Acknowledgements ................................................... vi
1. Introduction ...................................................... 1
   1.1 Area of research .............................................. 1
   1.2 Summary of work ............................................. 4
2. SLIM and the MC68000 ............................................ 6
   2.1 The SLIM Machine ............................................ 6
       2.1.1 The Registers ........................................... 7
       2.1.2 The Modifiers .......................................... 8
       2.1.3 A Summary of the Operations ........................ 11
   2.2 The Motorola MC68000 ....................................... 13
       2.2.1 The Registers .......................................... 13
       2.2.2 The Addressing Schemes ............................... 14
   2.3 Is the MC68000 suitable for SLIM? ........................ 16
3. The Mapping of SLIM to the MC68000 ............................ 19
   3.1 The Instruction Mappings ................................... 19
       3.1.1 Loading and Storing .................................. 19
       3.1.2 Sequencing ............................................. 23
       3.1.3 The Arithmetic and Logical Operations ............ 29
       3.1.4 The Miscellaneous Instructions ..................... 33
       3.1.5 Pseudo Directives .................................... 34
   3.2 Mapping Problems ........................................... 35
   3.3 Optimizations ............................................... 38
   3.4 The Runtime Support ....................................... 40
List of Figures

Figure 1. The SLIM Instructions ................. 12

Figure 2. The Calling Mechanism ............... 25
Acknowledgements

I would like to thank the department of Computer Science for its support on this project. This extends particularly to my supervisor John Peck, for his summer research position that got me started, and his unending support throughout. I also thank Harvey Abramson for reading this thesis, and for the BCPL version of the Call, upon which this implementation is based.

For the time I spent outside of the university, I thank Deborah. Her support saw me through the writing of this thesis.
Chapter 1
Introduction

The purpose of this endeavour is to study the feasibility of transporting the language BCPL to the Motorola MC68000 computer. This is to be accomplished through the intermediate language called SLIM, the assembly language for an ideal computer.

This allows a look at the concept of intermediate languages and their usefulness in the task of transporting languages. It specifically looks at the question of the use of SLIM for this purpose, as SLIM has been designed as an ideal computer for compiling BCPL programs into.

BCPL was chosen as it is a typical language in the field of computer science today. It is used for writing system software and user programs (Richards 69).

1.1 Area of research

The field in question deals with the idea of transporting languages to different computers by the use of an intermediate language. The use of an intermediate language gives the compiler writer greater freedom in code generation. This is because of the fact that there is one common standard, the intermediate language.
When a compiler writer is faced with the problem of code generation, there are two main options available to him. First, he may generate code that is highly dependent on the particular computer that he is using. Secondly, he may generate code that is for some ideal computer, and then worry about the specific problem of his actual computer.

Intermediate languages are also used in the area of transporting languages. Given a group of researchers on differing systems, three options are available for transferring programs.

The usual method is to transport programs at the source level. This is simple, fast, and guarantees that the program will work, assuming that the code has nothing that is machine dependent. The main problem is, what happens if researcher one has language X, and researcher two has language Y? This means that the code will have to be rewritten from X to Y. This can induce errors; for example, the original program may involve coding techniques that are unsupported in the second language.

The second method is the use of cross assemblers or cross compilers. This is having a program that translates the assembler language of machine X into that of machine Y. But again there are problems. The biggest is the overhead in writing these programs in the first place. There must be changes made at all of the installations if one of them
changes. Also there is a serious problem in the area of detecting bad translations. If my translation works in all but the case of the largest negative number, for example, I may never find this situation except after many runs of the program.

The last route that has been taken, is to define an intermediate language. This is usually the assembler language for some simple computer. The idea is that if I have this simple computer as hardware, then I can run any code that you send me in its assembly language. This can be accomplished by writing a cross assembler, or translator, that assembles this language into your native computer's assembler, or directly to code.

The fact that I now have only one translation to worry about at my installation has cut down the amount of code I must maintain, and it makes testing of the translation easier and safer. When I wish to transfer code to another installation, I only have to generate it for this intermediate assembler, move the source, and know that the code is correct for any of the systems.
1.2 Summary of work

The work that has been done can be summarized as follows:

- translate a BCPL program into SLIM,
- translate SLIM into Motorola MC68000 assembly code,
- assemble the code producing a Motorola S-module,
- load the S-module into a Motorola MC68000 simulator.

BCPL can be examined in several of the references. For an introduction, see "BCPL - The Language And Its Compiler" by Richards and Whitby-Strevens. SLIM and the Motorola MC68000 will both be discussed in the next chapter. The Motorola was chosen to be looked at as it is a typical modern micro computer, with a large set of instructions for use.

The translation of BCPL into SLIM is provided as one of the optional codes produced by the MTS/BCPL SLIM compiler. The MC68000 assembler and simulator are programs supplied by Motorola for developmental work. The area that I am involved in is the translation of SLIM into the Motorola assembler.

This entails several areas that have to be looked at. Firstly, is SLIM simple enough to be easily moved to a real computer? Secondly, is the MC68000 powerful enough to implement SLIM with any degree of efficiency? And thirdly, what mapping is necessary to equate the two computers?
The task is complicated by the fact that the Motorola software is written for an IBM-360. This is acceptable as UBC runs an Ahmdal-470, a 360 look-alike, but UBC also has an accounting system where every account is budgeted. The Motorola software is extremely expensive to run, and budgeting has been found to be the biggest problem. Therefore, it was decided that the developmental work would be done on a TI-990 computer, and once running, create the final version on the 470.

The TI system supports zed, a language that resembles C (Cheriton & Steeves 79). The translator was written and debugged in this, then it was translated into a BCPL version for MTS.

The savings taken by this method, made the difference between getting the implementation working or not. There can be a strong case made for not using simulators, but rather the real hardware.
This chapter has two purposes. It is to give a description of the SLIM and MC68000 computers. It is then to show that a mapping of SLIM to MC68000 is feasible.

It should be noted that the mapping is relatively simple and straightforward. There are, however, two major problems present. The first occurs in resolving the BCPL cell versus the Motorola word addressing schemes, whereby, Motorola means that a word address must be a 16 bit reference, occurring on an even address boundary. The other is that of resolving SLIM's upwards with the MC68000's downward growing stacks. These are discussed below.

2.1 The SLIM Machine

SLIM is a simple, one accumulator, stack oriented computer, which originated with the work of Mark Fox (Fox 78). It has since undergone several changes.

The biggest change, is that of compacting the notation (Abramson et al 80). This can be seen by examining Mark Fox's thesis's notation with the modern form. In the original, all of the instructions had a notation similar to most assembly languages. Now each instruction has a one or
two character notation, thus leading to a smaller number of bytes necessary to transmit source code from one installation to another. For example, the sequence that was LD P7 STKLD 0 PUSH is now represented as LIE7 PL0 P.

Another main area of change has been in the call and return mechanisms. Originally, the routine that was to do the call, had to reserve sufficient space on the stack for the environment. Now the call instruction handles that job as well. The tradeoff made was one of fewer instructions but with a more complicated call.

2.1.1 The Registers

SLIM is designed to work with a runtime evaluation stack. To accomplish this, there are two registers used for stack maintenance.

The first is called E and is used to define the environment. All of the runtime linkage, parameter, and local variables are accessed as offsets from E. An offset is defined in the BCPL sense, as the number of cells, not the number of words or bytes.

The second register used with the stack is H. This is the top of stack marker (also known as the high point in BCPL literature) and is used for pushing and popping values
to and from the stack. It can be dynamically changed during execution, thus allowing blocks of cells to be allocated or removed from the stack in addition to 'one at a time' access.

For computation, there is an accumulator cell called $A$. All arithmetic and logical operations take place using $A$. By convention, this is also used for holding the value returned by a function.

The last two registers are $G$ and $C$. The concept of the BCPL global vector is carried through to SLIM with $G$. It points to the start of the global area. At the present time this is set by the program initialization, and is not alterable at runtime. SLIM's program counter is $C$.

2.1.2 The Modifiers

In the SLIM computer, many of the operations take a possibly modified operand. This may be thought of as a value that is used in some way by that operation. It should be noted, however, that there are operations that do not take an operand, and that there are several forms of modifiers.

If an operation, say $L$, uses a modifier, the value that $L$ uses is computed before any action is taken by $L$. This
value, after modification, is called W. W may be thought of as another register, one that is set to a value immediately after the next instruction has been fetched. It is W that the operation then uses.

Every operand is either a signed or unsigned number, a character constant (this is stored as an integer value), or a label. One of these forms must be present if the operation takes an operator. The notation for a number is just a numerical value, preceded by a '-' sign if negative. For a character constant, the character ' is placed immediately in front of the character that is the operand. For a label the modifier '@' is used. This is followed by the label number. One of the above forms is always the last in an instruction that takes an operand.

The number (character, label) may be immediately preceded by either an 'E' or a 'G'. They may not be both present in one instruction. The 'E' means that the current value of the E register is to be added to the operand producing the modified value, thus implementing a mechanism to give displacements in the runtime environment stack. Similarly, 'G' means to add to the operand the value of the G register. This gives an offset into the global area. At present, G's value never changes, but is assumed to be set to an area that represents the globals.
At this point, \( W \) has a number. This is just a value, but there must be a mechanism to get at the value stored in some location. This could be accomplished by having an instruction that does this task; load from location. This was not the chosen route. It was decided that any instruction should have this ability, not only to handle values, but values in locations. This is represented by following the instruction opcode with an optional 'I'. If there, it means that there is to be one level of indirection taken on \( W \) at this point.

As an example, take the instruction LIE-5. First, \( W \) is loaded with the value -5. Then the current value of the \( E \) register is added to that. What you now have is a pointer to the runtime environment, -5 cells from the current environment pointed to by \( E \). The 'I' next indicates that you should treat this as an address and fetch the value at that location. So \( W \) is now loaded with the contents of the cell that is pointed to -5 cells from the current location of \( E \). Finally, it is this value that is used by the instruction, L (load the accumulator with \( W \)).

There is a totally different operand that may be used. It is represented by the character 'H'. It may not be used with any of the others already mentioned. What it will do is to take the present value of the \( H \) register, use it as an address, load \( W \) with the value that \( H \) points to, and then decrement \( H \). This is the action of popping a value off of
the runtime evaluation stack. Note that the fact that $W$ is set before any action of an instruction must be taken into account with this modifier. Some instructions expect a modified value and an additional value from the stack, so ordering is critical.

2.1.3 A Summary of the Operations

The operations supported by SLIM, fall into a number of categories. The first is that of loading and storing. These are always to and from the accumulator, thus setting up arithmetic sequences and storing answers, and for moving data from one area to another, as when setting up the parameters for a procedure call. Another instruction type is that of sequencing. There are basically the jump (conditional and unconditional), and the call and return mechanism in this category, but also include sequential switching controls. Next, fall the arithmetic and logical operations. Most of these take a modifier; all affect the accumulator. Finally there are a number of miscellaneous instructions. A summary of the instructions of SLIM is provided.
The SLIM Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mnemonic</th>
<th>Microcode</th>
<th>Note: D = data</th>
<th>@ = label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load cell</td>
<td>LW</td>
<td>A := W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store cell</td>
<td>SW</td>
<td>!W := A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load cell subcripted</td>
<td>L!W</td>
<td>A := A!W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store cell subcripted</td>
<td>S!W</td>
<td>LET V = !H; H := 1; A!W := V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load byte</td>
<td>L%W</td>
<td>A := A%W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store byte</td>
<td>S%W</td>
<td>LET V = !H; H := 1; A%W := V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load field ***</td>
<td>L:W</td>
<td>A := W of A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store field ***</td>
<td>S:W</td>
<td>LET V = !H; H := 1; W OF A := V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load device ***</td>
<td>L$r</td>
<td>A := A!r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store device ***</td>
<td>S$r</td>
<td>LET V = !H; H := 1; A!r := V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Push and load cell</td>
<td>PLW</td>
<td>H +=: 1; !H := A; A := W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jump</td>
<td>JW</td>
<td>C := W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>True jump</td>
<td>TW</td>
<td>IF A = TRUE THEN C := W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>False jump</td>
<td>FW</td>
<td>IF A = FALSE THEN C := W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modify high point</td>
<td>MW</td>
<td>H +=: W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;op&gt; ****</td>
<td>&lt;op&gt;W</td>
<td>A := A &lt;op&gt; W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call **</td>
<td>CW</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return **</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Push</td>
<td>P</td>
<td>H +=: 1; !H := A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exchange</td>
<td>X</td>
<td>W := H!(¬1); H!(¬1) := A; A := W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complement</td>
<td>¬</td>
<td>A := ¬A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negate</td>
<td>¬</td>
<td>A := ¬A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute</td>
<td></td>
<td></td>
<td>A := ABS A</td>
<td></td>
</tr>
<tr>
<td>Originate ***</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Void ***</td>
<td>V</td>
<td>no operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Switchon sequential *</td>
<td>?S</td>
<td>D@default Dc1 Dc1 Dc2 D2 ... Dcn D@n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Switchon indexed *</td>
<td>?I</td>
<td>D@default Dlo Dhi D@1 D@2 ... D@n</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* See "The Essence Of Portable Programming" (Abramson et al 80) for the microcode

** See "Yet Another Call And Return together with Revised Remember and Transfer" (Abramson 81) for the BCPL version of the code

*** Not implemented

**** <op> codes are: + - * / /* < <= > >= = ¬ = >> << == ¬ / \ - /
2.2 The Motorola MC68000

The Motorola MC68000 is one of the new generation of micro computers. It has an internal data size of 16 bits, with a 23 bit address space (word addressable). The instruction set is very full for arithmetic and control operations.

2.2.1 The Registers

There are 8 data, and 8 address registers. Each of these is 32 bits in size. The machine supports operations in byte (8 bit), word (16 bit), and long word (32 bit) for the data registers. The address registers may only be used in word and long word size.

When dealing with byte size, the low order (right hand) 8 bits are the ones that are affected. The same is true with the word size. It is the low order 16 bits that are affected. When long word is chosen, all 32 bits are affected.

Any of the data registers may be used in computation. There is no default register, the programmer has complete control. This is also true for the address registers, except for the action of a procedure call. Address register 7, A7, is used for this purpose. It actually is two
different registers, determined by the mode of the machine.

In user mode, A7 is the user stack pointer. The MC68000 also supports a system mode, however. When the computer is in this mode, A7 is the supervisor stack pointer. This project does not consider the use of the supervisor mode of operation, so any procedure call will use the user stack pointer.

2.2.2 The Addressing Schemes

The MC68000 supports addressing schemes that reassemble those of the PDP-11 series. There are twelve different addressing schemes. Some of the instructions have implicit register references. The call and return is an example of this.

A list of the various schemes is:

- data register direct

- address register direct

- address register indirect

    treat the value in an address register as an address
-address register indirect with postincrement

  the value of the increment is adjusted according to the size of the argument (1, 2, or 4)

-address register indirect with predecrement

  the value is again adjusted according to the size

-address register with displacement

  add a displacement value to the contents of the register

-address register with index

  in addition to a displacement, add another register's value into the address calculation

-absolute short

  the address is contained in the next word

-absolute long

  the address is contained in the next two words

-program counter with displacement

  as with address register with displacement

-program counter with index
as with address register with index

-immediate

the data is in the instruction, for byte and word size the next word contains the value, for long word, a second word is used

2.3 Is the MC68000 suitable for SLIM?

The point to consider is whether or not the two machines are similar enough to try to make a mapping. The philosophy of SLIM is one of simplicity, so that mapping to any real computer is possible. The mapping is feasible, and is relatively simple. I will now show that SLIM can be mapped to the MC68000.

The SLIM accumulator, A, is handled by the data register D0. The program counter C is handled by the MC68000's own program counter. The address registers 0 and 1 are used for H and E respectively. Register A4 is used for G, although for this implementation, it is set to 0.

Given that the MC68000 supports 16 bit operations easily on data, 16 bits was chosen to be the cell size. This means that the maximum addressability is only 16 bits, thus cutting down on the actually available number, but the
simulator used to test the system only allowed addresses in the range of 0-7FFF hex. This meant that a cell could contain the address of any addressable word in the simulator.

The main reason for a 16 bit cell, was that the hardware easily supports this for the arithmetic and logical operations. A major problem is that the MC68000 only uses the low order 23 bits for addressing. This would imply that a 23 bit cell would be the largest that could be supported, as BCPL (and SLIM) state that a cell must contain an address. Motorola has plans to expand the computer to a full 32 bit address space in the future, so a 32 bit cell is possible for future implementations.

At the present time, A4 is initialized to 0. This places the global vector at that location. The maximum number of globals that has been implemented is 25. The address of the startup routine is placed at location 0 (that is, at global 0), the default address that the simulator starts at.

For communication, the program uses two simulator supplied functions, INCON and OUTCON. These are input from console, and output to console, respectively. They deal with a record at a time, so the BCPL routines, RDCH and WRCH are defined in terms of these. This makes line by line I/O simple.
With the above, it has been seen that every component in SLIM can be equated to the MC68000. The mapping will therefore be complete if the instructions can also be mapped. This is the topic of the next chapter.
Chapter 3
The Mapping of SLIM to the MC68000

Given that the structure of SLIM has been mapped to the MC68000, there are three things that must be seen. The first are the instructions that will do the mapping from SLIM to the MC68000. The other two involve problems with that mapping. The first of these is to resolve the problem of SLIM's use of an address as a cell address with that of the MC68000's use as a word address. Lastly, SLIM's stacks are defined to grow upwards. This is inconsistent with the MC68000, whose stacks grow more conveniently downwards.

3.1 The Instruction Mappings

This section looks at each instruction group in turn. This allows one to concentrate on the specifics of a particular set of instructions.

3.1.1 Loading and Storing

SLIM supports 5 different load and store instruction pairs. They always load or store in or out of the accumulator. The various instructions are:
Given that the load and store device has not been finalized in the design of SLIM, and that loading and storing a field can be accomplished by logical operations, it was decided not to do these. It should be noted that loading and storing are not consistent in their interpretation of W. For loading, W represents the value that is to be loaded into A. But for storing, W represents the address that A's value is to be placed in. This corresponds to the inconsistent actions of the left and right hand side of an assignment statement in most programming languages.

For simple loading and storing, just the value is needed for A. To handle cells, an index is needed. A problem occurs for loading bytes. That is the fact that loading a byte into a data register in the MC68000, will only alter the low order 8 bits. This means that for a BCPL/SLIM sense, the other byte of the low order word of A must be cleared. The upper two bytes are not affected, as they are only used for addresses.
The next area that must be looked at is how to handle the various modifier options. The method chosen, was to have the code as orthogonal as possible, taking advantage of the various addressing schemes supported by the machine, rather than emphasizing different instruction sequences.

To handle no modifier is just to supply the data as being of an immediate nature. If the modifier is E or G, the SLIM address in those registers must be supplied.

There is a slight complication though, the addresses stored in the E and G registers (on the MC68000) are stored as word addresses, thus allowing indexing to take place easily. This means that to get the cell address, it is necessary to make an adjustment. This is accomplished by taking the value and shifting that right by 1 bit.

The only place where this shifting must not be done, is in the global loading sequence, because if an address is in the global vector, it must be a word address, not a cell address. For the IE and IG modifiers, all that is needed is to convert the cell address into a word displacement. This is just twice the cell value, taking appropriate action for the direction of stack growth. If the modifier is H, the operand is just popped off of the stack. A label has its address loaded, shifting to a cell address if necessary. An indirect label is handled with the program counter with displacement mode. Finally, for a character, the machine
dependent form is loaded into the low order byte.

For storing, the scheme is similar, except that the address is taken one level sooner; that is, it handles the inconsistent nature of storing. This is apparent in examining E and IE. For the E modifier, this is a location that is some displacement from the E value. But for IE, the value at that location must be loaded, and then use that for the address to place A.

Examples:

<table>
<thead>
<tr>
<th>Example</th>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-456</td>
<td>MOVE.W #-456,D0</td>
<td>Load a constant</td>
</tr>
<tr>
<td>LE3</td>
<td>MOVE A1,D0 LSR.W #1,D0 SUBQ.W #3,D0</td>
<td>Load a cell address Note the correction for the address</td>
</tr>
<tr>
<td>LH</td>
<td>MOVE.W (A0)+,D0</td>
<td>Pop the stack</td>
</tr>
<tr>
<td>SE2</td>
<td>MOVE.W D0,-4(A1)</td>
<td>Store at a location</td>
</tr>
<tr>
<td>SH</td>
<td>MOVEA.W (A0)+,A2 MOVE.W D0,(A2)</td>
<td>Store with top of stack indirection</td>
</tr>
<tr>
<td>L!4</td>
<td>MOVE.W D0,A2 MOVE.W -8(A2,A2),D0</td>
<td>Load with a cell subscripted</td>
</tr>
<tr>
<td>L%IE6</td>
<td>LSL.W #1,D0 MOVEA.W -12(A1),A2 MOVE.B 0(A2,D0),D0 ANDI.W #$FF,D0</td>
<td>Load a byte The upper order byte of the accumulator is cleared</td>
</tr>
</tbody>
</table>
Also to consider in this group, are the actions of P and PL. They represent pushing onto the evaluation stack. For the PL instruction, there is the additional action of loading A after the push has occurred.

The code for both is:

```
MOVE.W D0, -(A0)
```

This is followed by the standard load sequence for PL.

3.1.2 Sequencing

There are three methods of sequence control in SLIM. They are branching, the calling and returning pair, and the switchon constructs.

SLIM supports both unconditional and conditional branching. For unconditional, a jump to the label is provided. For conditional, a test is made with A. Depending on the outcome, a branch may occur. For an indirect label, the value at the specified location is loaded into an address register, then that is used as the
SLIM uses the BCPL/MTS sense of true and false. True is represented by negative 1. False is represented by 0.

SLIM does not state what happens if A has a value of other than true or false, and a T or F instruction is executed. With this implementation, you may not branch with the T, whereas you would with the F, and vice versa.

To see this, suppose that A has value -2. If you have the series T@4 J@5, the T instruction will compare -2 with true. Since -2 is not equal to true, the instruction will not jump, therefore causing the J@5 to be executed. However, if you have the series F@5 J@4 (which logically should branch to the same label as the previous series), -2 is not equal to false so the branch will be to label 4.

To see how call and return work, the structure of the runtime stack is necessary (see figure 2). Basically it consists of 3 cells that contain in order, the old top of stack pointer, the old environment pointer, and the return address. For a call to occur, all of the parameters, except the last to the routine, are loaded on the stack. The last is loaded in A. The call instruction treats W as the address of the routine.
The Call In Action

Before the call

After the call

Figure 2
The Calling Mechanism
To handle the case of differing numbers of formal and actual parameters, this information is explicitly coded into SLIM. A call of C@3 D4, is a call to the routine at label 3, with 4 actual parameters. Then elsewhere may be @3:D6, meaning that this routine expected 6 formal parameters. The purpose of this information is so that the environment can be set up in accordance with the compiled offsets, as these can not be determined at run time.

The return instruction, R, restores H, E, and C from the 3 cells currently being pointed to by E.

Because of the complexities of calling a routine, the C and D instruction pair was made into a runtime support procedure. The address of the routine to call is loaded into A2, with the number of parameters passed, loaded into D1.

Switchons have two forms, sequential, and indexed. These were both considered too complex to generate the code for directly, so they are implemented as runtime routines. The code sequences for these are as defined in SLIM. There is the instruction to jump to the runtime support routine, followed by the various data declarations. These declarations are not handled in any special way, but are just translated as they are.
The runtime support for SLIM's call instruction, must have the address of the call loaded into A2, with the number of parameters passed loaded into D1. The routine will then look at the SLIM routine's number of expected arguments, and will construct a new environment. It will then pass control to that routine.

The return instruction will simply restore the old E and H pointers, and transfer control to the return address.

Examples: (assume the first section)

```
C@3
D3
LEA L3S1,A2
MOVEQ #3,D1
JSR CALL

CI@4
D2
MOVEA L4S1,A2
MOVEQ #2,D1
JSR CALL

CH
D0
MOVEA.W (A0)+,A2
MOVEQ #0,D1
JSR CALL

R
MOVEA.W A1,A3
MOVEM.W (A3)+,A2/A1/A0
JMP (A2)
```

For an unconditional branch, a jump to the label is all that is needed. If it is a conditional branch, however, a test with the accumulator is needed first. This will consist of either looking for -1 or 0 (true, false) and jumping if the test succeeds.
Examples: (assume the first section)

\begin{verbatim}
J@10  JMP L10S1
JI@10 MOVEA.W L10S1,A2
        JMP (A2)
T@1  CMPI.W #-1,D0
        BEQ L1S1
F@2  TST.W D0
        BEQ L2S1
\end{verbatim}

The two switchon instructions are implemented as runtime support routines. The mechanism for translation is simply to place a jump to subroutine, JSR, instruction to the appropriate routine. The SLIM data following it is simply translated as normal.

Examples: (assume the first section)

\begin{verbatim}
?S  JSR SSW
D@1  DC.W L1S1
D1  DC.W 1
D@2  DC.W L2S1
D10  DC.W 10
D@3  DC.W L3S1
D100  DC.W 100
D@4  DC.W L4S1

?I  JSR ISW
D@1  DC.W L1S1
D10  DC.W 10
D12  DC.W 12
D@2  DC.W L2S1
D@3  DC.W L3S1
D@4  DC.W L4S1
\end{verbatim}
3.1.3 The Arithmetic and Logical Operations

The chief reason for choosing a cell size of 16 bits was because of this group of instructions. The MC68000 supports the concept of a 16 bit data size for the arithmetic instructions. Simple arithmetic, such as addition and subtraction, is possible at a size of 32 bits, but multiplication and division are not. For ease of programming, these two were programmed using the machine's own multiply and divide.

SLIM does not state what happens on overflow, underflow, or on divide by zero. The results of these occurrences are undefined and strictly dependent on the particular computer that SLIM is implemented on. For this implementation, there is no detection done, so runtime errors will result.

The arithmetic operations that are supported are addition, subtraction, multiplication, division, and modulus (in the BCPL sense for signs). All of the above are for integer arithmetic only, as currently there are no floating point numbers implemented in SLIM.

The relational operators set the value of A, after doing a specific comparison of A to some value. Those supported are less than, less than or equal to, greater than, greater than or equal to, equal, not equal. The value of A will be set to negative 1 if the comparison is true, 0 if it is
There is also the operations of shifting. Left and right are both supported. The bit that is shifted in is value 0. Note that SLIM does not have a rotate operator.

The accumulator may also be used for bit manipulation, not just comparison. The operations supported are equivalent, not equivalent, logical and, and, logical or. The bit pattern formed by the current value of A applied to W is left in A.

There are a number of operations that do not take a modifier. These are bit wise complement, negate, and absolute value. All of these affect the accumulator.

All of the above operations have a machine hardware equivalent, with the exception of absolute value. This was handled as a stand alone runtime routine.

For addition and subtraction, the value is taken and applied to D0 directly. It should be noted that there is no checking for underflow or overflow. This is in agreement with what SLIM states.

For division and remainder, the machine's own hardware divide is used. It expects a 32 bit number, to be divided by a 16 bit number, and will return the signed integer
division and the remainder. To generate the 32 bit dividend, a sign extension is performed. For a /* instruction, the division is performed, then the remainder is swapped into the low order word.

Examples: (with an unmodified operand)

+1 ADDI.W #1,D0
-2 SUBI.W #2,D0
*3 MULS #3,D0
/4 EXT.L D0
   DIVS #4,D0
/*5 EXT.L D0
   DIVS #5,D0
   SWAP D0

For relational operators, the comparison is made with the value to be tested against. At this point, the MC68000's condition register is set. Now the set instruction is used. It will set the byte destination specified if the given condition is true, otherwise, clear it. To make this a SLIM true or false, a sign extension is then done.

Shifting is done using one of two forms. The first is for shifts of from 1 to 8 bits inclusive. The other is for bigger shifts, this one requiring that a second data register contain the number of bits to shift.
Bitwise comparisons are handled as expected, using the MC68000's own hardware instructions.

The only complication of those instructions that have no operand, is with absolute value. This is not a hardware instruction for the MC68000. So it must be done in software. The complication is that the assembler does not support a reference to the current location. This means that if a test for being negative is done, a branch around a negate instruction cannot be done. The ABS routine was therefore created.

Examples: (with an unmodified operand)

```
<6  CMPI.W #6,D0
    SLT D0
    EXT.W D0

>=7  CMPI.W #7,D0
    SGE D0
    EXT.W D0

>>8  LSR.W #8,D0

<<9  MOVEQ #9,D1
    LSL D1,D0

==10  EORI.W #10,D0

<=11  EORI.W #11,D0
    NOT.W D0

\12  ANDI.W #12,D0

7    NOT.W D0
```
3.1.4 The Miscellaneous Instructions

As mentioned earlier, SLIM can modify the H register dynamically, not just when pushing or popping the stack. This is by the M instruction. The value of W is directly added to H, in a cell addressing sense.

One instruction is provided for easing the job of the compiler writer. This is the top of stack exchange, X. The action of this is to exchange the accumulator with the contents of the cell that is one down from the top of the evaluation stack. So if A has value 1, the top of stack 2, and the cell below the top of stack 3, after executing X, A will be 3, the top of stack 2, and the other cell 1. This operation requires 5 instructions on the machine, but occurs so infrequently that it is left as a straight in code insertion, not a runtime routine.

Examples:

M5 SUBA.W #10,A0
M5 SUBA.W 4(A1),A0
M5 SUBA.W 4(A1),D0
3.1.5 Pseudo Directives

There are a number of pseudo directives that SLIM has. These are used for storing constants, defining labels, defining the start of procedures and sections, and telling when a global load sequence starts.

All of these are handled by using similar pseudo directives of the Motorola assembler. A point to note is that it is the $$ directive that directs the status of shifting or not for adjusting for cell addresses.

SLIM generates label numbers starting at 1 for each section, thus having labels unique for each section. The Motorola assembler does not support mixing of assemblies. The method chosen for making the labels unique, was to append an additional 'S' followed by a number that represented the section number. So SLIM's @10 in the first section will translate to L10S1, and in the fourth section to L10S4. This makes the labels unique for the assembler.
Examples: (assume the first section)

@2: L2S1:

D1 DC.W 1

D@5 DC.W L5S1

D'C DC.W <Character representation of C>

$"START" ***** START

$ * procedure end.

These 3 give directions to the translator:

$$"FIRST" ***** Section: FIRST

$$ <start of global load sequence>

. <end of section>

3.2 Mapping Problems

As mentioned before, there are two main problems facing the mapping. The first of these is the fact that SLIM uses a cell address and the MC68000 uses a word address (which is twice that of the cell address). The second is that of configuring the translation to fit SLIM's upwards growing stack, with the MC68000's downward growing stack.
In BCPL, if I say \( X := @Y \), I will have the cell address of \( Y \) loaded in \( X \). This concept is carried over to the SLIM machine, so that if the accumulator has an address loaded into it, it will be the cell address. This means that for any other instruction that uses this value as an address, it must double that value before use. If it does not replace the accumulator's value, it must in addition reset that value back to a cell address, in case the next operation uses the accumulator again with a cell interpretation. This may occur, for example, if there are two store cells in a row.

The only time when you do not want to do this, is in the global load sequence. This can be shown if you were to load global 15 with \( @1 \). Now the compiler generated code will reference this in the manner of an indirect call through global 15. A typical call might be, CIG15 D0, so the value address. On the other hand, if I need to do some arithmetic to a variable that has an address, it must be at the cell level so that it conforms with the BCPL/SLIM sense. This can be seen by examining the code for Writef. In this routine, you have a variable in which you decrement by 1 after each use, thus giving you a cell address.

The other problem of the stack growth is relatively easy to handle. The MC68000 does handle stack growth in both directions, but is simpler in the downward direction. This
fact can be seen by remembering that the addressing schemes are predecrement, and postincrement. This means that for forward growing stacks, you would be pointing one word beyond your top of stack word.

The choice was made to utilize the more natural (for the MC68000) downward stacks. This made the problem transparent to SLIM except for the instructions that have an address and use a SLIM displacement from that. An example of this is L!3. This tells SLIM to take the value in A, treat it as a cell address, and get the cell that is 3 locations up the stack from this. Handling the cell address has been discussed above, but for the 3 locations beyond that, you have to treat this as a cell displacement and double it.

Finally, a look at vectors, strings, and addresses is required. The vectors are handled in the compilation of the SLIM code as a compiler option. For upward growing stacks, the cell pointing to the vector is made to point to the first cell of that vector. This means that the cell indexed by v!n is n cells up from @v. This would not work for downward growing stacks, however, as v!1 is 1 cell down in the runtime stack. To solve this, the cell pointing to the vector is made to point to the last cell of the vector. This will cause the vector to grow 'upwards' in the runtime stack, so v!n is n cells 'down' the stack, but is n cells into the vector. In effect, the directional sense of the vector has been reversed. Strings are stored in the code
generated, which grows upwards, so they are not a problem as they go the correct direction already. As for addresses, since the MC68000 uses downward stacks, all references to the 'next' BCPL or SLIM cell must use -1 as the increment. For example, if X has the address of some parameter, to get the address of the next parameter, you would write X -:= 1.

3.3 Optimizations

SLIM is described as a simple, one accumulator, stack oriented computer. The simplicity can best be seen by how the BCPL code for X := X + 1 would typically be generated. It would look like: LIE3 +1 SE3.

From this it should be apparent that at least as far as incrementing or decrementing goes, SLIM uses three instructions where one would do. But this would also be true for any updating type assignment, for example, X := X + Y, which would give LIE3 +IE4 SE3.

The reasoning is to keep the number of SLIM instructions to a minimum. The above would have to have a dyadic operation. This proliferation of additional operators would defeat the purpose in having a simple computer.

The above is just part of the picture, however. The problem shows up in another example.
The code for \texttt{LIE3 =4 \texttt{T@5}}, would create 6 MC68000 assembler instructions, where 3 would do. The problem can best be seen in that the individual SLIM instructions carry no information to the next instruction. This would alleviate such problems as the conditional branch example.

If such information is allowed to be carried across more than one SLIM instruction, then particularly efficient instructions for your machine may be taken advantage of.

This was recognized and dealt with in this implementation. The way to do this is to see that instruction 1 followed by instruction 2, can best be done, not necessarily by directly translating them in turn, but possibly by transforming them into a different pair of instructions. The way to accomplish this, for the particular example of loading a series of parameters to the stack before calling, was to invent a new instruction; load to stack, \texttt{LS}.

To see this, consider the BCPL statement \texttt{foo(1,2,3)}. This will generate the SLIM code \texttt{L1 PL2 PL3 CI@3 D3}, where you have \texttt{@3:D\textless\label of foo\textgreater}. The loading series will generate 5 assembler instructions. With the optimization this will be seen by the code generator as \texttt{LS1 LS2 L3 CI@3 D3}. A single \texttt{LSn} will take only one assembler instruction, so the loading series will take only 3 assembler instructions. So typically, you will go from \(2 \times (n-1) + 1\) (n
is the number of parameters passed), to $n$ instructions. This is a considerable savings no matter how you measure efficiency.

The fundamental conclusion in experimenting with optimizing the code generation, is that the implementor must choose a fixed number of SLIM instructions to look at and then potentially transform these into a new set of instructions that take advantage of the particular computer.

3.4 The Runtime Support

For any computer system to run, there must be some runtime support. For this system, it consists of a library of BCPL functions, and some primitives to perform certain SLIM instructions.

The format of the primitives, is just to load any control information that they require, and then generate a jump to subroutine instruction, JSR. The most frequently used primitive, is the CALL routine. This is for implementing SLIM calls. It functions by taking an address, and the number of actual parameters, and will then set up the runtime environment (depending on the actual number and the formal number of parameters), and then transfer control to the code.
An example of the functions available in the library is RDCH. This will return with the next character from the current input line. If there is no character, it will get the next line. Note that for this implementation, there is a newline character, *N, at the end of every line.

The last area that can be included as runtime support, is that of the loading phase. For this system, this is a static phase, as the global area and library are preassembled, and start at a fixed location (0). The program starts assembling at the next available location beyond the library. The code of the program is automatically assembled with a sequence of instructions that load the address from global 1, and begin execution there. This is followed by a series that tell the user that the program execution is finished, and ask for termination.
When considering a project of this type, the first question that can be asked is, does it work? There are other factors than this, however. Questions such as the size of code produced, and the speed at which that code will run may be just as important.

To answer the first question, does it work, one only has to look at the results of some runs (see appendices). The answer is yes it runs. A BCPL program can be taken and compiled to SLIM. The SLIM can be translated to Motorola MC68000 assembler and successfully assembled producing a Motorola S-module (this is the term for a loadable module to the Motorola developmental system, of which the simulator used in this project is a part). This S-module can be loaded and run.

One of the other factors is the size of the code that is produced. It may mean the difference between a program that will fit on the machine and run and one that won't. For the present simulator, the total of the program code, the runtime library, the global area, and the runtime stack must be less than 8000 hex words. This is not an overly generous amount of memory if one wishes to implement a compiler, but this is sufficient for small to moderate programs.
If you have a system in which you want to implement a larger program, there are four routes that are available. The first is the easiest and is to purchase a real machine with sufficient memory to fit the program you have in mind. To do this you must remember that the code generator must be reworked to handle the fact that there will be sign extension when dealing with addresses for the MC68000. The usual route is to hardwire the address lines from the CPU to low, thus ignoring the problem.

The second is to implement the large program in pieces that are of manageable size (will fit), and have multiple passes. With this scheme each pass must have a representation of its data that will be storable on the mass storage device. This may not be too practical for some structures; for example, the expression tree that is produced by the BCPL compiler. This is not impossible to represent on a disk, but is easier to handle if it is all internally stored.

Alternately, the software may page the data structures to and from the mass storage device. But again, a representation of the data must be chosen that makes this feasible.

The last method is the most expensive of the four in terms of hardware costs, but is the most flexible. It is to combine the first two, and add the memory management unit
that Motorola manufactures. This will allow you to implement a virtual memory system. This would alleviate the problem of insufficient memory and allow the software not to have to do the paging.

As far as execution speed is concerned, this project cannot make an empirical measurement, as there is no access to a real MC68000, only to a simulator. The simulator will not execute in real time, given that it is on a time sharing system. One can not even make a good guess at the real time, as the system load will be shifting over time.

The only way to measure, is to use the simulator's ability to set break traps. This will allow you to see the number of machine (simulated) cycles that a program takes. This is illustrated in the appendices.

The biggest problem with this system, is that the code produced is not the most compact form that is possible. This is caused by the fact that SLIM's principle of being a simple computer does not take into account the fact that the newer micro computers have powerful new features. An example of this is the link mechanism for the MC68000.

This instruction will set up a stack link, reserve a given number of words and store the old link pointer, thus allowing the machine to keep its own form of an environment, at the instruction level. Unfortunately, this is
incompatible with the design of SLIM, because SLIM also needs to keep the top of stack marker along with the environment pointer. This results in the route that was taken, a library primitive that does the actions of the SLIM call.

The code that is produced must work in the general case, therefore, it does things that it sometimes does not need to. An example of this is in handling the cell to word address problem.

Many times, the value in the accumulator is shifted left by 1 bit, thereby making it a word address. After this interpretation has been finished, the accumulator's contents are then shifted right by 1 bit, thereby maintaining its cell address form. This is the general case, as mentioned. The problem is that frequently the accumulator is then immediately used for some other purpose, overwriting the value that was just corrected. An optimization would be to notice that the value in the accumulator is to be overwritten and not make the shift.

This problem of not producing the most compactful code was most noticed when the library was hand optimized. The primitive library can be (and is) defined in BCPL, with the exceptions of Rdch, and Wrch. This was compiled producing a SLIM'ed version of the routines. This was then translated to form the assembler code. The assembler was not directly
assembled, however.

At this point, the assembly code was taken and modified. The first thing that was done was to take the code and install into it, the area that was for the global vector. At location 0, that is, global 0, a single word form of a branch was installed to the Startup code. This is the section of code that sets up the machine's stack, the SLIM stack, initializes the registers used for I/O, and then branches to the standard starting address.

Once this step had been taken and found to be working, the individual routines were taken, one at a time, and were examined and hand optimized, to reduce the space they occupied and their execution speed. The non-optimized versions worked, but the tradeoff of the manual effort for the improved performance was deemed valid. The underlying conclusion of this is that there must be another pass made over the code, one that does some data flow analysis (possibly having to make a tree, in addition to the original expression tree that was made to produce the SLIM in the first place).

The results of hand optimizing the code, show that the code produced is correct, but that it is not near to optimal. This can be seen to be a fault, not of the code generator, but rather of the lack of specialization in the definition of SLIM. The SLIM philosophy does not preach
compact native code, it just states simplicity of machine design, and ease of implementing. Both of these points have been shown to be true by the results of this project.

The work also shows that for a particular implementation, a local compiler writer may have two versions of his compiler. The first is for the original problem of transporting code to a different installation. For this, a well defined common design is needed. For other work, however, he may well want to have a SLIM prime machine defined for his computer. Think of the SLIM prime as a SLIM computer, but with extras added that take advantage of information he knows that he will be able to use at code generation time. It would be a combination of this and the translation of SLIM sequences to differing sequences, that would make for much better code.

A specific example was made with the MC68000. The load to stack instruction, LS, as previously mentioned, was created. Its purpose is to take advantage of the fact that the MC68000 can move data directly to the runtime stack, it does not need to load a register and then push the register's value.
Chapter 5
Conclusions and Further Research

The intention of the project was to see whether SLIM could be a suitable vehicle for the transporting of BCPL programs to the Motorola MC68000. To this end, the project is a success. The code exists which will take SLIM code and produce Motorola assembler which will load and execute on the Motorola supplied simulator.

There are problems as things stand, though. The first and foremost is that this is using a simulator, not a real machine. This was caused by several factors, principally that the university does not yet own a MC68000. The second factor, is that even though the university has purchased a MC68000, it will not be available in the near future. This does not invalidate the project, it just would have been nicer to see the code executing on a real machine.

A topic that must be looked at before research on SLIM can proceed, is whether or not it should be redesigned to take advantage of new features of real micro computers. This is a debatable topic. There is the argument that the efficiency that this would buy is well worth the effort and complications that this would add. This would be reflected in better code generation (or at least no needed code optimization), and in smaller source code programs (they
would be more compact).

Of course, the argument against this is that SLIM must be simple, so that anybody has a chance of implementing it. This is the argument that I tend to support. The simpler the simulated computer, the easier my initial job is. This leads to a better chance of my getting a system that will run.

At many installations, code size and execution speed are critical issues. These may be caused by memory size limitations, non-mapped memory (a no virtual memory system), and real time applications.

Even with these problems, however, it is better to get a translator which works before handling writing an efficient translator. Given that an implementor is functioning from within a community of users, someone may already have written an efficient translator for his particular machine. If this is so, he need only have a simple minded translator running to get the efficient one.

He can get the person with the efficient translator to SLIM it, and send the SLIM'ed code to him. He then runs the SLIM'ed efficient translator through his not so efficient one. Then he runs the SLIM through the code that was produced, giving him a native machine version of the efficient translator.
If an efficient translator does not exist, and he writes one, then it is available in the community of users. Thus, if someone has the same machine as he does, the above actions are reversed. Another person writes a simple minded translator, and then is sent a SLIM'ed version of the efficient one.

Even if other users may not be able to use the efficient translator for their machines, it may provide some methods that they can use themselves. In effect, what an efficient translator for a particular machine can do, is to provide a framework on which another can be built. This may take the form, for example, of a new instruction (for the MC68000, the LS is an example) which another machine can use.

If the route followed is to keep SLIM as simple as it is, then an implementor must have a method for the optimization, as mentioned above. This can be done by combining two methods, optimize a series of SLIM instructions into a different series producing assembler code followed by optimizing that assembler code, before the actual assembly.

It is this combination that must be looked at for future machines. The question of what is a good instruction to invent for one machine, for example the LS (load to stack) for the MC68000, may well be asked on another machine.
The LS instruction was designed for the MC68000 to take advantage of its stack nature. The same instruction would probably be used on a PDP-11, as the MC68000's addressing schemes are based on the PDP's. This would indicate to an implementor, that if his machine is stack oriented, he may well be able to make use of LS.

After this comes the question of how to take advantage of the fact that most modern computers have instructions that contain two memory references, so operations like incrementing, do not have to do a load, add, store sequence, but only 1 instruction.

Looking at this closer, a typical SLIM sequence is: L1E5 +3 SE5. This would produce the code:

```
MOVE.W -10(A1),D0
ADDI.W #3,D0
MOVE.W D0,-10(A1)
```

These three instructions can be reduced to one having the same effect:

```
ADDI.W #3,-10(A1)
```
Bibliography


2. Abramson, H. Yet another Call and Return! together with Revised Remember and Transfer, UBC SLIM note, 1981


10. MC68000, 16-Bit Microprocessor, Motorola, 1980


Appendix I

The Software

This is a description of the software involved with this system. This includes the BCPL to SLIM compiler, the SLIM to Motorola translator, the MC68000 assembler, and the MC68000 simulator.

The BCPL to SLIM compiler is a program that originated at Cambridge. It has since been modified at UBC to run under MTS. This is part of the ongoing SLIM project. BCPL has been successfully compiled on several different computers. The route taken has been one that this thesis has done, convert BCPL into SLIM and then write a SLIM to native computer translator. To facilitate the fact that some computers have a downward growing stack (like the MC68000), the compiler has an option that will correctly handle VEC's, a BCPL one dimensional array construct, to take this into account.

The translator will accept SLIM code and produce Motorola MC68000 assembly code. Each instruction is parsed, to get the instruction, the modifier, and the operand. The operator is then used to switch into a large SWITCHON statement, which then in turn, uses the modifier (if necessary) in another SWITCHON. The code is then generated by the final state that is found. The program also uses the
simple optimization of examining two consecutive SLIM instructions, and replacing them with a different pair, if they are found to be a known pairing.

The Motorola assembler and simulator are programs supplied from Motorola. The assembler will generate a text file that is acceptable by the simulator. These files are called S-modules, as every line starts with an S.

All of the above software can be seen running in the following appendices. This is a list of the programs:

-To generate SLIM code:
  \$RUN JELP:SLIM SCARDS=BCPL.source SPUNCH=slim PAR=BN

-To translate this into assembler:
  \$RUN RNGR:PROJ.O+CS:BCPLLIBLE SCARDS=slim SPUNCH=asmb

-This is then given to the assembler:
  \$RUN VERX:M68K.ASM SCARDS=asmb SPRINT=-P SPUNCH=s.module

-This is then run on the simulator:
  \$RUN VERX:M68K.SIM 1=s.module+RNGR:PROJ.LIB
Appendix II

Print Out Hi

```
# com This is an edited version of the programs
# com running under MTS
#
# list t1.s
> 1 // test WRCH
> 2 get. "rngr:header"
> 3 let start() be
> 4 $( Wrch('H'); Wrch('i'); Wrch(' N') $)
# com Now compile this into SLIM
# run jelp:slim scards=t1.s spunch=t1.sl par=bn
UBC - BCPL/MTS version 7 (81-09-20)
PAR=BN
(S) listing = FALSE
(T) tree = FALSE
(n) code option =
(B) backvec = TRUE
(L) lex trace = FALSE
(O) overlay = FALSE
(N) naming = FALSE
(K) mark = FALSE
(+) tree space = 12000
Tree size = 426(000652)
0 BCPL errors detected

# list t1.sl
> 1 J@2
> 2 "$START"
> 3 @1:D0 L'H CIG2 D1 L'i CIG2 D1 L'*N CIG2
> 4 D1 R $: $
> 5 @2:@3:$$ L@1 SG1 .
> 6
# com Now translate this into Motorola assembler
#
# run proj:o+cs:bcpllib scards=t1.sl spunch=t1.a
Motorola instructions = 21(000025)

# list t1.a@-ic
> 1 $CONTINUE WITH RNGR:PROJ.HD.S RETURN
> 2 JMP L2S1
> 3 ***** START
> 4 L1S1: DC.W 0
> 5 MOVEQ #72,D0
> 6 MOVEA.W 4(A4),A2
> 7 MOVEQ #1,D1
> 8 JSR CALL
> 9 MOVEQ #105,D0
> 10 MOVEA.W 4(A4),A2
> 11 MOVEQ #1,D1
> 12 JSR CALL
> 13 MOVEQ #10,D0
> 14 MOVEA.W 4(A4),A2
> 15 MOVEQ #1,D1
```
> 16    JSR CALL
> 17    MOVEA.W A1,A3
> 18    MOVEM.W (A3)+,A2/A1/A0
> 19    JMP (A2)
> 20    * procedure end
> 21    * procedure end
> 22    L2S1: DS.W 0
> 23    L3S1: LEA L1S1,A2
> 24    MOVE.W A2,D0
> 25    MOVE.W D0,2(A4)
> 26    $CONTINUE WITH RNGR:PROJ.TL.S RETURN

# com This is now assembled, producing an S-module
# run verx:m68k.asm scards=t1.a sprint=-p spunch=t1.68
# MC68000 ASM REV: 1.4 - COPYRIGHT MOTOROLA 1978

***** TOTAL ERRORS: 0 -- TOTAL LINES: 49

# list t1.68
> 1    S006000004844521B
> 2    S11305324EFA00300007048346C000472014EB868
> 3    S1130542004E7069346C000472014EB8004E700A99
> 4    S1130552346C000472014EB8004E36494C9B0700BD
> 5    S11305624ED273
> 6    S113056445FAFFD0300A39400002346C00027200AC
> 7    S1130574EB8004E700A45F8007272014EB8004E2F
> 8    S11305844E7200004BFA00184DED001B4EB90001E9
> 9    S11305940004EB9000100004EB900010000202AF7
> 10   S11305A42A2A20454E4204F462052554E202A2ABA
> 11   S10D05B42A2048495420454F4620F0
> 12   S9030000FC

# com Now this load module and the library
# com are loaded into the simulator
# run verx:m68k.sim 1=t1.68+proj.lib
MACSS SIMULATOR - REL 1.9
? IM
? SD T
? R
Hi
PRIVILEGED INSTRUCTION TRAP ADDRESS IS ZERO.
T=3C6
end of file
# com Note that there were 3C6 Hex cycles taken
Appendix III

Towers of Hanoi

# com This is a simple example that will
# com demonstrate recursion and
# com multiple sections
#
# list t15.s
> 1  section. "first section, just has start"
> 2  get. "rngr:header"
> 3  global $( hanoi : 15 $)
> 4  let start() be
> 5  $( let n = readn()
> 6  if n = 0 then return
> 7  writef("Towers of Hanoi n=%N*N", n)
> 8  hanoi(n, 'A', 'B', 'C')
> 9  $) repeat
> 10
> 11
> 12
> 13  section. "second section"
> 14  get. "rngr:header"
> 15  global $( hanoi : 15 $)
> 16  let hanoi(n, s, i, d) be
> 17  $( if n <= 0 then return
> 18  hanoi(n-1, s, d, i)
> 19  writef("Move %N from %C to %C*N", n, s, d)
> 20  hanoi(n-1, i, s, d)
> 21  $)
> 22
> 23

# run jelp:slim scards=t15.s spunch=t15.sl par=bn
UBC - BCPL/MTS version 7 (81-09-20)
PAR=BN
(S) listing = FALSE
(T) tree = FALSE
(n) code option =
(B) backvec = TRUE
(L) lex trace = FALSE
(O) overlay = FALSE
(N) naming = FALSE
(K) mark = FALSE
(+) tree space = 12000
SECTION first section, just has start
Tree size = 479(000761)
0 BCPL errors detected
SECTION second section
Tree size = 525(001015)
0 BCPL errors detected

# list t15.sl
> 2  $$"first section, just has start" J@2
> 3  "$START"
> 4  @1:D0
> 5  @3:CI G8 D0 P =0 F@4 R
> 6  @4:L@5 PLIE1 CI G5 D2 LIE1 PL'A PL'B PL'C CI G15
Towers of Hanoi

Move 1 from A to C
Move 2 from A to B
Move 1 from C to B
Move 3 from A to C
Move 1 from B to A
Move 2 from B to C
Move 1 from A to C

Towers of Hanoi n=1
Move 1 from A to C

0

PRIVILEGED INSTRUCTION TRAP ADDRESS IS ZERO.
T=1123E
end of file

# com Note the number of cycles
Appendix IV

The SLIM Parser

# com This example takes the
# com SLIM parser (modified)
# com from the program
#
# com It is then run on the SLIM
# com that writes out 'Hi'
# com (see appendix 2)
#
# run jelp:slim scards=eg.s spunch=eg.sl par=bn
UBC - BCPL/MTS version 7 (81-09-20)
PAR=BN
(S) listing = FALSE
(T) tree = FALSE
(n) code option =
(B) backvec = TRUE
(L) lex trace = FALSE
(O) overlay = FALSE
(N) naming = FALSE
(K) mark = FALSE
(+) tree space = 12000
Tree size = 2855(005447)
0 BCPL errors detected
# run proj.o+cs:bcpllib scards=eg.sl spunch=eg.a
Motorola instructions = 877(001555)
# run verx:m68k.asm scards=eg.a sprint=-p spunch=eg.68
MC68000 ASM REV: 1.4 - COPYRIGHT MOTOROLA 1978

****** TOTAL ERRORS: 0 -- TOTAL LINES: 930
# run verx:m68k.sim 1=eg.68+proj.lib
MACSS SIMULATOR - REL 1.9
? IM
? SD T
? R
$continue with t1.sl return
code=162,7,2
code=179,12,16243
code=166,9,1
code=152,9,0
code=165,11,72
code=151,4,2
code=152,9,1
code=165,11,105
code=151,4,2
code=152,9,1
code=165,11,10
code=151,4,2
code=152,9,1
code=182,13,0
code=179,13,0
code=179,13,0
code=166,9,2
code=166,9,3
code=1,13,0
code=165,7,1
code=183,3,1

PRIVILEGED INSTRUCTION TRAP ADDRESS IS ZERO.
T=3460D
end of file

# com You can now compare this with the
# com manifest values for the instructions
# list eg.header
> 1  MANIFEST $( cell_bsz = 2; endstreamch = -1 $)
> 2  /* The tree header. */
> 3  MANIFEST
> 4  $( s_section = 1
> 5      s_neg = 2
> 6      s_not = 3
> 7      s_mult = 5
> 8      s_div = 6
> 9      s_rem = 7
>10      s_plus = 8
>11      s_minus = 9
>12      s_eq = 10
>13      s_ne = 11
>14      s_lt = 12
>15      s_ge = 13
>16      s_gt = 14
>17      s_le = 15
>18      s_shl = 16
>19      s_shr = 17
>20      s_logand = 18
>21      s_logor = 19
>22      s_neqv = 20
>23      s_eqv = 24
>24      s_abs = 36
>25      s_c = 151
>26      s_d = 152
>27      s_end = 153
>28      s_ent_def = 154
>29      s_error = 156
>30      s_ext_def = 157
>31      s_f = 158
>32      s_isw = 161
>33      s_j = 162
>34      s_l = 165
>35      s_lab_def = 166
>36      s_load_byte = 168
>37      s_load_cell = 169
>38      s_m = 173
>39      s_p = 177
>40      s_pl = 178
>41      s_proc_nm = 179
>42      s_q = 181
>43      s_r = 182
>44      s_s = 183
>45      s_ssw = 185
MANIFEST

/* These are constants for SLIM modifiers. */
$( m_e = 1; m_ie = 2; m_g = 3; m_ig = 4
  m_h = 5; m_ih = 6; m_l = 7; m_ill = 8
  m_n = 9; m_in = 10; m_c = 11; m_s = 12
  m_null = 13; m_colon = 14; m_en_def = 16;
  m_ex_def = 17
  m nop = 18
  */ States for each external */
  ext_declared = 1; ext_defined = 2;
ext_applied = 3 $
$)

GLOBAL $( section_num : 15 $)
This example will show that though the code produced works, it could be improved by having another pass over it. This pass could do simple data flow analysis, for instance, it could determine if the accumulator is to be loaded with a new value in the next instruction. If so, then it is free to be bypassed in a load sequence.

This is seen if you consider that a typical SLIM sequence is to load the A with a value, then store it somewhere, then load another value. This could instead be accomplished by just using an instruction that will move the first data item to where it is needed, and then loading in the next value.

This is the code produced by the program for Writes, the standard BCPL routine for writing strings. This is one of the library routines that is supplied.

**** WRITES
L3S1: DC.W 1
  MOVE.W #1,-(A0)
  MOVE.W 6(A1),D0
  MOVEA.W D0,A2
  CLR.W D0
  MOVE.B 0(A2,A2),D0
  MOVE.W D0,-(A0)
  JMP L24S1
L25S1: MOVE.W 6(A1),D0
  LSL.W #1,D0
  MOVEA.W -2(A1),A2
  MOVE.B 0(A2,D0),D0
  ANDI.W #$FF,D0
This code was then taken and hand optimized. This was done for space and time efficiency in the library routine.

***** WRITES
Writes: DC.W 1
MOVE.W #1,-(A0)
MOVEA.W 6(A1),A2
CLR.W D0
MOVE.B 0(A2,A2),D0
MOVE.W D0,-(A0)
BRA.S L24S1
L25S1: MOVE.W 6(A1),D0
LSL.W #1,D0
MOVEA.W -2(A1),A2
MOVE.B 0(A2,D0),D0
ANDI.W #$FF,D0
MOVEA.W 4(A4),A2
MOVEQ #1,D1
JSR CALL
ADDQ #1,-2(A1)
L24A1: MOVE.W -2(A1),D0
CMP.W -4(A1),D0
BLE L25S1
MOVEA.W A1,A3
MOVEM.W (A3)+,A2/A1/A0
JMP (A2)
* procedure end
* procedure end
The non-optimized version has 27 instructions and one constant. The other has 21 and a constant. Even this simple example shows that the code can be compressed.

This optimization might be done by reading the assembler instructions in order, looking for a time that the accumulator (DO in this system) is loaded with a value. The scan then continues to the following instruction. If that instruction is to store DO, the next instruction is examined. If this one is to load DO, then the previous two can be changed from the load DO, store DO, into move a value.

This can be seen by examining:

```
MOVE.W #2,DO
MOVE.W D0,2(A1)
MOVE.W -6(A1),D0
MOVE.W D0,8(A4)
MOVE.W -4(A1),D0
JMP L1S1
```

This can be changed into:

```
MOVE.W #2,2(A1)
MOVE.W -6(A1),8(A4)
MOVE.W -4(A1),D0
JMP L1S1
```
Appendix VI

The CALL Routine

This is the CALL routine, as it is in the runtime support library. It is based on the BCPL routine in "Yet another Call and Return ! together with Revised Remember and Transfer" (Abramson 81).

This BCPL version was run through the SLIM compiler, hand assembled from the SLIM produced, and finally, hand optimized. This is the most compact code that I can see for the routine.

CALL: MOVE.W (A2),D2,               \ get number expected
      BEQ.S CALL1                             \ if −= 0 then
      MOVE.W D0,-(A0)                           \ push last argument
CALL1: MOVE.W D2,D4                       \ if indefinite number
      BLT.S CALL2                             \ then skip this:
      SUB.W D1,D4                              \ compute new_h(Shortfall)
      LSL.W #1,D4
      SUBA D4,A0
      MOVE.W D2,D1

CALL2: LSL.W #1,D1                        \ adjust H
      MOVE.W A0,D2
      ADD.W D1,D2
      MOVEA.L (A7)+,A3                      \ get return address
      MOVEM.W D2/A1/A3,-(A0)               \ make stack environment
      MOVEA.L A0,A1                         \ E := H
      ADDQ #2,A2                              \ skip over number expected cell
      JMP (A2)                               \ start executing the routine