Technical Report

UCAM-CL-TR-129
ISSN 1476-2986

Number 129

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

A methodology for automated design of computer instruction sets

Jeremy Peter Bennett

March 1988

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted January 1987 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Emmanuel College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

Series editor: Markus Kuhn

ISSN 1476-2986

Contents

Preface

With semiconductor technology providing scope for increasingly complex computer architectures there is a need more than ever to rationalise the methodology behind computer design. In the 1970's byte stream architectures offered a rationalisation of computer design well suited to microcoded hardware. In the 1980's RISC technology has emerged to simplify computer design and permit full advantage to be taken of very large scale integration. However such approaches achieve their aims by simplifying the problem to a level where it is within the comprehension of a single human being. Such an approach is not sufficient. There is a need to provide a methodology that takes the burden of design detail away from the human designer, leaving him free to cope with the underlying principles involved.

In this dissertation I present a methodology for the design of computer instruction sets that is capable of automation in large part, removing the drudgery of individual instruction selection. The methodology does not remove the need for the designer's skill, but rather allows precise refinement of his ideas to obtain an optimal instruction set.

In developing this methodology a number of pieces of software have been designed and implemented. Compilers have been written to generate trial instruction sets. An instruction set generator program has been written and the instruction sets it proposes evaluated. Finally a prototype language for instruction set design has been devised and implemented.

The first chapter is a survey of instruction set design in its historical context. The next three chapters describe the development of the new design methodology and its evaluation. The final chapter surveys the theoretical basis of the methodology in an attempt to discover ways of improving instruction set design still further. No part of this work was done in collaboration, nor has any of it been, nor is any of it currently being submitted for any degree, diploma or other qualification of any other university.

I should like to thank my supervisor, Dr Martin Richards for his advice and encouragement over the last three years. My thanks also go to my colleagues in the School of Mathematics at Bath University; Professor James Davenport for endless advice and perceptive criticism, Dr. Geoff Smith for teaching me the method of Lagrange Multipliers and the importance of mathematical rigour and most important of all Professor John ffitch who has acted *in loco supervisoris* with great vigour for the past three months. During the period of this work I was supported by a Research Studentship from the Science and Engineering Research Council.

1. Analysis of the Problem

"The value of a megabyte of memory strongly depends on the computer in which it is installed." [Wirth86].

The field of computer design is broad. It encompasses the work of the electrical engineer, the computer theoretician, and the language designer. Any innovation in this field must take account of a substantial body of existing work in a variety of fields. The first part of this chapter examines the problems involved in instruction set design, and introduces the particular problems with which this dissertation is concerned. The remainder is devoted to a survey of work done in this field, with detailed examination of particularly relevant work.

1.1. The Problem

The instruction set of a computer represents the interface between the ideas expressible in a high level language, and the functionality that can be provided by electronic hardware. With the

development of microcode as a design tool [Wilkes51] it has become relatively simple to adjust the instruction set to "improve" the quality of this interface.

The question arises of what is meant by an improvement in quality. One measure might be the absolute speed of compiled programs. Another might be the size of the compiled code. However it is notoriously difficult to pin down what part of any improvement is due to change in the interface. Part of any change must be attributed to alterations in the hardware such as increase or decrease in gate delays. With microcoded architectures some benefit may be due to programming tricks possible with particular instructions. Part of any gain must be attributed to alterations in the compiler generating the instructions. Particular optimisation techniques may be possible with one instruction set and not with another. Enabling such optimisations to be used is all part of good instruction set design, but after such improvements have been allowed for what remains can be fairly said to be improvement due to better design of the interface. For practical purposes the distinction may be irrelevant. If an instruction set design methodology causes an improvement it is a useful methodology, whatever the mechanism. However to a scientist attempting to analyse the underlying behaviour, in the hope of improving the methodology, it is vital that the source of the improvement be identified and characterised.

The term *architecture* and the phrase *instruction set* are used rather loosely throughout the literature, with a variety of shades of meaning. Throughout this dissertation they are used with precise meaning. *Instruction set* refers to the interface presented to the compiler writer as his sole means of controlling the actions of the computer. *Architecture* means the complete hardware of the computer supporting the instruction set, together with the instruction set itself.

There is one method of instruction set design in which performance improvements can be attributed to the change in interface alone. This is when an instruction set is extended by combining or splitting existing instructions or providing special cases of existing instructions. There is no modification to the compiler, since the new instructions can be implemented by peephole substitution of the existing instruction set as a post-compilation operation. Furthermore if we use criteria of improvement that are not dependent on the hardware or

microcode implementation, such as static or dynamic code size[1] we can eliminate benefits due to hardware or microcode changes. This dissertation concerns itself with instruction set design using peephole optimisation. In chapter two the benefits that can be gained from minimising static code size are investigated. Much of the work, in particular the design of an instruction set for BCPL described in chapter three, will aim to minimise static code size. In the same chapter application of the approach to other languages is demonstrated and a generalisation to support multiple languages and different design criteria is proposed and discussed.

The difference between the ideas expressed in high level language constructs and the ideas implemented in low level hardware is termed the *semantic gap* and must be bridged by the instruction set. It would seem reasonable to suppose that an ideal instruction set should reflect concisely both sets of ideas, introducing as little redundant information as possible in describing the algorithm written in a high level language to the hardware. How closely to either side of the semantic gap an instruction set may lie is not clear. There have been a range of views expressed by machine designers, from the approach used by SYMBOL [Rice71], whose instruction set is the high level language itself, to machines such as EDVAC and RISC-1 [Patterson82], whose instruction sets reflect closely the hardware. Where to stand along this spectrum is unclear, and has changed over the years. The history of this change is described in the rest of this chapter.

Currently two main strategies have evolved to bridge the semantic gap between high level language and hardware. The first strategy attempts to provide relatively high level instructions, that reflect the essential operations of a high level language. Such instructions perform complex operations on multiple operands that can be addressed in many ways. There are models of referencing that allow for indirection and indexing and flow of control operations that understand procedure calls and stack frames. Such machines are usually implemented as microcoded architectures. Microcode is itself an interface to the low level operations of the hardware, but this

---

[1] *Static Code Size* is used to mean the amount of space occupied by the compiled code. Clearly it is advantageous to minimise this to obtain machines that need less main memory. *Dynamic Code Size* is used to mean the amount of space occupied by the compiled code, weighted by the frequency with which each instruction is executed. Clearly decreasing this will help reduce memory-processor bandwidth.

aspect of interface design is beyond the scope of this dissertation[2]. Architectures such as these must support more than one language. This has often been done by making the high level instructions more general in their specification. Such generalisation is not always successful, for example the NS32000 series [National83] clearly supports Modula2, for which it was originally designed, better than LISP or BCPL [Wilson83]. Recently there has been some development of soft microcoded architectures which permit changing of instruction sets on the fly [HLH85]. This offers one way out of this problem. Computers with architectures of this general type are known as *Complex Instruction Set Computers* (CISC's).

Critics point out that to support many high level languages the instructions of a CISC must be very general in their operation and parameterisation. As a result they are slower and larger than needed for any one specific language. A classic example is seen in the implementation of Portable Standard LISP [Griss82] on the VAX-11. Function calling does not make use of the VAX-11 general purpose CALL opcode, as is the case with Franz LISP [Foderaro80], but uses a sequence of simpler instructions that execute faster [Davenport83]. The alternative of instruction sets that are more language specific makes the machine less general, and provides the compiler writer with a heavy burden in generating code to use the instructions provided. A solution to these problems that has come to prominence recently is the *Reduced Instruction Set Computer* (RISC) [Patterson80].

RISC's have a only a small number of very simple instructions, typically memory-register transfer, register-register arithmetic and conditional branching. However the very simplicity means that the compiler writer can select precisely the sequence of instructions he needs to translate a particular high level construct, instead of searching, possibly unsuccessfully for a complex instruction to perform the operation. By this means a precise semantic match is obtained, and the amount of redundant information generated in the process of compilation minimised. At first sight it would seem such processors would lose by virtue of having to execute so many more instructions. However since there are so few instructions and they are so very simple, a great deal of effort can be put into making them execute at high speed. The

[2] For a general discussion of systems as multiple layers of interfaces see [Tanenbaum84].

processor can often be implemented directly in hardware, removing the overhead introduced by the microengine. The processors that are now appearing certainly seem to bear this out, with extremely good benchmark results. For example the Acorn RISC [Wilson86] runs BCPL at about half the speed of an IBM 3081/D and LISP rather faster than a GEC 63.

RISC's do have some problems. Their code is both statically and dynamically large, giving extensive memory requirements, and heavy loading of the memory-processor bus. The Acorn RISC produces BCPL code that is 1.2 times the size of code for the MC68000 and Modula2 code that is twice the size. It also has to use a 32 bit data bus and paged mode access to memory to obtain the requisite 20Mbyte/second bandwidth required by the processor. Under these conditions the fastest DRAM chips are operating at full speed. There is little scope for increasing the speed of the CPU (as is proposed in the second version of the processor), without radical redesign of the interface to memory. The fact that these overheads are not larger is a reflection of the poor quality of CISC instruction sets, and the number of extra instructions that have to be generated because of inexact mapping of high level constructs to complex instructions. RISC's have also failed to show the expected language independence. Invariably the simple instructions, and underlying architecture are better suited to some languages than others. RISC-1 [Patterson82] is very much a machine for running C, and much of its gain can be attributed to an original view of how C should be supported, rather than the specific simplicity of its instruction set.

The argument over the relative merits of these two approaches has raged for some time, and is as much about laboratory rivalry as computer science. A good survey of the debate can be found in [Colwell85]. This dissertation attempts to combine the benefits of both design methodologies to give processors with the compactness and light bus loading of CISC, but with the speed and simplicity of RISC. To achieve this compromise we must look first at the evolution of computer design and some historical landmarks along the way.

1.2. Chronology

The key influence in opening up the field of computer design was the invention of microcode [Wilkes51]. It was a simple mechanism for providing a specialised architecture on a general machine. Although there was research into the influence of programming language use on computer design as early as 1953 [Hopper53], it was not until technology advanced sufficiently for microcoding to become a practical implementation technique that widespread experimentation with the theory of computer architecture design became feasible. It is interesting to see that the advances in computer design that have resulted from research and experimentation over the past thirty years have moved us towards computers in which microcode is very little used, RISC machines.

It would be inappropriate to provide a comprehensive survey of the whole of computer architecture within the context of this dissertation. There is an excellent survey of computer architecture in general from the earliest days of computer design in [Siewiorek82]. This dissertation concerns itself only with ideas germane to instruction set design for computers that support high level languages. Supplementary material on early work with direct execution architectures, particularly SYMBOL is to be found in [Chu75].

In the earliest computers the instruction set was a direct reflection of all the facilities the computer provided. In effect the semantic gap did not exist. The early high level languages were designed initially to support existing hardware, and so again there was little incentive to provide specialist hardware. The three dimensional limit on FORTRAN-66 arrays had more to do with indexing on the IBM 709 than language design considerations. However with high level languages came a degree of abstraction and separation of software from the constraints of hardware. Experience of designing algorithms in high level languages, devoid from such restrictions, led to the problem being turned round. Algol60, COBOL and LISP represent a development of languages influenced by the needs of the programmer, rather than the facilities of the hardware (as was the case with autocodes and to a large extent FORTRAN). With the development of these languages, support in hardware was then required for the abstract concepts they had introduced.

The Burroughs B5500 [Lonergan61] introduced a number of ideas to support high level languages. It introduced the idea of a hardware stack for intermediate results during expressions. It also introduced the idea of instructions composed of a number of symbols to designate format opcode and operand information. The IBM 1401 and later IBM S/360 introduced character operations to support COBOL. Although in the light of modern developments in hardware support for software these developments may seem small help, they were significant enough for Rosen to suggest in 1968 [Rosen68] that

> "All the features that have been designed into digital computers may be considered as reflections of software needs."

Such innovations, for all their originality and long term significance, were still in essence minor manipulations of existing ideas on how hardware should be built. In parallel with these efforts a number of far more radical designs were under development. These *direct execution architectures* were the first attempt to drive computer design purely from the perspective of the programmer.

1.3. Direct Execution Architectures

The earliest workers were constrained by current technology to look at imaginary or "gedanken" machines. The first approach used was to try to execute the textual source code directly. Anderson [Anderson61] looked at the design of an extension to the B5500 that would execute Algol-60 directly. It ran into the problem of dealing with operator precedence, and required a second operator stack to resolve this problem. Other designs (for example SYMBOL, described later in this section) introduced pre-processing into polish form to eliminate this problem.

There were a number of other gedanken machines during the 1960's which looked at the idea of supporting a high level language directly. Perhaps the most sophisticated analysis was by Bashkov [Bashkov67]. This followed a microprogrammed design proposed in outline by Melbourn and Pugmire [Melbourn65]. Bashkov proposed a machine to execute FORTRAN programs, with minimal compilation. He was working with an early version of FORTRAN without subroutines, yet his design still required an estimated 10 000 semiconductors to implement. It

was essentially a hardware implementation of a one pass load and go compiler. The only translation performed before execution was the conversion of labels and variables to addresses, everything else was kept in raw textual form. Bashkov suggested 4K words of memory would be adequate for such a machine.

An example of a real machine following Bashkov's way of thinking was developed in the SYMBOL IIR project, in the five years to 1971 [Rice71]. This provided a hardware compiler for the SYMBOL Programming Language (SPL), capable of translating 70 000 - 100 000 statements a second. SPL is a high level procedural language in the Algol-60 mould. It is translated by hardware into a reverse polish string of operation codes and operand addresses for execution. A virtual memory system is provided, allocating and claiming back memory as necessary. The various units that make up the machine were kept separate, to allow maximum parallel processing. It made use of early integrated circuits, and needed a total of 68 printed circuit boards using 18000 SSI chips. Both this and Bashkov's machine moved away from the von Neumann architecture that had become standard and returned to the separate code and data spaces of the Harvard Mark I. The SYMBOL computer was a one off machine, and although used for a while in research never fully met its design objective of demonstrating that hardware implementation of languages would lead to better computer systems. Reviewing the project ten years on the system designer expressed some frustration with the extent of the machine's success [Rice81].

These designs represented a switch from one side of the semantic gap to the other, with instruction sets that were more or less identical to the high level language. Indeed they are hardly instruction sets in the conventional meaning of the term. Euler was implemented as a microcoded interpreter on the IBM S/360-30 [Weber67]. It represented a step away from the compiler completely implemented in hardware, translating the source code with a microprogram into a reverse Polish notation, more suited to hardware execution. A similar approach was reported by several workers in microcoded translation of APL to a functional form for execution, for example versions by Zaks in 1971 and Hassit in 1973 described in [Chu75]. Further evolution away from the hardware compiler was suggested by Shapiro [Shapiro72] in his SNOBOL machine. SNOBOL [Griswold71] is a language for manipulating data in string form, based on

Markov algorithms. Shapiro proposes a conventional von Neumann machine with additions to support twin stacks, character data and pattern matching. At the time SNOBOL was implemented by interpreting an intermediate code, OCODE, and such additions to a machine would have given great improvement in performance. The development of the SNOBOL compilers, SPITBOL and Macro-SPITBOL [Dewar83], with an observed ten fold increase in machine performance have rather undermined Shapiro's work, and indicate the danger of ignoring gains to be made by improving software.

Although very much in the direct execution tradition Shapiro's suggestions for modification of existing machines reflected a change in direction. The idea of a conventional machine that could be configured to support a variety of language systems as needed, by microcoding of a very general microengine, was perhaps centered on the development of the Burroughs B1700 [Wilner72]. The underlying machine was designed to support interpretation of whatever intermediate machine was appropriate. Its use of the bit, rather than byte or word, as the basic unit of information gave considerable scope for experimentation with instruction sets to support particular languages.

1.4. Analysis of Language Usage

These early designs were very much influenced by what designers felt ought to be in a language orientated machine. It was only at the start of the 1970's that people started to analyse the use of existing machines, to provide data with which to design future machines. The first work of this sort was a comparative analysis of FORTRAN programs from industry and commerce [Knuth71]. Knuth examined 440 programs, representing over 250000 lines of code from Lockheed computer division and 11000 lines of code, representing rather more than 440 programs from Stanford undergraduates. The interesting points were not in the comparison of style, but in the exposure of some unexpected details of language usage. Knuth discovered that the average expression has only two operands, indicating that support for complex expression evaluation is perhaps unjustified. Notable is the fact that a large proportion of expressions are of the form x+1 or y*2. The fact that the commonest statement is assignment is perhaps reassuring, since load and store operations are invariably well supported, particularly since 68%

also turned out to be simple repacements of the form A=B. The small size of loops (only 13% containing more than five statements) would indicate that relative branches are a necessity, particularly when modern machines require 24 or more bits for an absolute address operand. The high frequency of particular operators (addition and multiplication) suggests a need for specialist support. Knuth also distinguishes between static and dynamic statistics, showing for the first time that optimisation of program size and program speed require different concepts. His dynamic profile of 24 programs shows surprising similarity in frequencies of instructions, with the exception of assignment (51% static, 67% dynamic) and DO loops (9% static, 3% dynamic). The fall in DO loop count is to be expected, since we clearly execute statements inside the DO loop more than the DO loop itself. The rise in the proportion of assignment statements would be an indication that assignment is a common occurrence within loops.

Since Knuth's original work analysis of high level language usage has become something of a popular pastime. Weicker published a summary of sixteen such analyses [Weicker84]. The quality of these analyses varies considerably, and in general reflect greater ease with which static, rather than dynamic statistics may be collected. Weicker uses the statistics to derive a program which has a typical static and dynamic distribution of constructs. The results are in general agreement. Static results indicate assignment is the commonest statement, occurring between 34% and 54% of the time, with the lower figures being for the block structured languages. Procedure call is the next commonest statement throughout, although with a wider range of results from 12% in Knuth's analysis of FORTRAN to 40% in Shimasaki's analysis of Pascal. Here there was a clear distinction between languages, Pascal in the range 29%-40%, Ada in the range 24%-27%, XPL and PL/I in the range 15%-17%. This shows clearly how particular languages emphasise particular programming styles. IF statements of various forms were next most common, accounting for 9%-18% of statements. GOTO is infrequent except in FORTRAN (9%) and PL/I (3.8%), and a good predictor of the date of a language's introduction. Looping constructs other than FOR loops occur with a frequency roughly inversely proportional to the frequency of GOTO statements. The dynamic data is less complete, but in general shows the same result as Knuth, that is an increase in proportion of assignment statements and a decrease in proportion of looping constructs. A summary of the left and right hand sides of assignment statements is given. Statically, assignment to a simple variable is commonest,

accounting for between 56% and 71% (it should be pointed out that there are only two analyses) of all such statements. Assignment to structures is much rarer. The fairly obvious result that assignment to array elements is much more common on a dynamic rather than static count is given by [Grune79]. The right hand of an assignment is commonly a constant, occurring in 25%-44% of cases on a static count, and a slightly lower proportion reported for a dynamic count. Components of a structure or array elements form the other common simple right hand part of assignment statements. Between 19% and 24% of right hand sides are expressions on a static count, with Tanenbaum reporting a dynamic count of 33.5% [Tanenbaum78]. Static counts suggest that 2% to 5% of right hand sides are expressions with more than one operator, although Tanenbaum's dynamic statistics suggest this rises to 13%. Addition and subtraction dominate the dynamic and static statistics for arithmetic operators, although multiplication is significant for FORTRAN and Algol68. Of the relational operators equality and inequality dominate. Logical operations are relatively infrequent (2%-14%). Figures are also given for the number of parameters in procedure calls. The two sets of figures for static counts suggest an approximately exponential decay in frequency of occurrence against number of parameters, with 41%-47% having no parameters, 19%-31% having one, 14%-15% having two and so on down to between 1% and 8% having five or more parameters. Tanenbaum's dynamic statistics suggest that zero, one and two parameters are approximately equally common, with a fall off for larger numbers of parameters. Operands are shown to be mostly simple integer variables. Figures are given for the locality of variables. There is considerable variation from analysis to analysis, but most fall into the category local or global, with local variables being more common on dynamic analysis, and global on static analysis.

Weicker's work is an excellent summary and analysis of the work to date, as has been shown by the great amount of interest in his "Dhrystone" Benchmark, which attempts to behave as a characteristic program. The volume of data is considerable, but it is essential that any language or machine designer read and digest it thoroughly before proceeding.

Bashkov's machine and SYMBOL IIR are both computers that attempt to eliminate any need for compilers, and are truly machines that support directly executed languages. However high level languages are designed to be written and read by human beings, and there is no reason why

they are necessarily the best interface to low level computer hardware. Following on from hardware compilers have come machines with instruction sets that are a compromise between the hardware and the software.

During the 1960's machines had started to acquire features under the influence of high level languages. The IBM S/360 is a prime example, drawing on earlier innovations in IBM design. It provided floating point operations to support FORTRAN and character operations to support COBOL. The PDP-11 range drew on ideas introduced by the B5500 with its hardware stack. These were all features provided by hardware innovators, whose background was in general electronic engineering.

Much work, including this dissertation, has gone into looking at what is the best representation of a high level language program for execution. The combination of machines such as the B1700 that were easy to reconfigure and statistical analyses like that of Knuth gave rise to a new breed of machine designers, who rather than being electronic engineers had a background in compiler writing.

1.5. Byte Stream Architectures

Rather more innovative machines resulted from compiler writers who turned their hand to machine design. The MESA instruction set for the Alto [Johnsson82] was an early example, and the same approach was used for CINTCODE [Richards84]. The designers looked at operations in existing machines which had proved most useful to them in existing instruction sets, and produced machines that contained just those features, with support given to each feature according to its importance. These machines all take a very simple view of an instruction set as a single operation code, one byte long, with a number of arguments following after. This is a development of the syllable structure of the B5500 architecture, and is in essence the approach used by the IBM S/360 architecture. The instruction set is a prefix polish notation. Unlike other architectures there is in general no format syllable, with the format being implied from the opcode byte. Such instruction sets are simple to implement (at least at the conceptual level) as table lookup on the opcode byte. Such an approach is well suited to microcode, with

microengines that automatically switch on the opcode byte. The size of one byte for an opcode need not limit the number of instructions, since one or more of the opcodes can be "escape" opcodes, with the next byte taken as a subsidiary opcode[3]. Architectures with instruction sets of this type are known as *byte stream architectures*, and are sufficiently important that three examples will be looked at in more detail, to study the ideas that have evolved during their development.

CINTCODE is a byte stream instruction set to support the BCPL programming language on machines with 16-bit addressing. BCPL is a simple block-structured typeless language [see Richards80 for a description]. It was not designed to be implemented in hardware, but to be interpreted in software on small machines. The aim was to permit large BCPL programs to be run on small machines, and compactness of compiled code was a major design criterion. CINTCODE as published only supports 16 bit BCPL, appropriate given the size of machine it is to run on. However 32 bit versions have been developed both by Padget and ffitch [Padget83] and for the purpose of this dissertation. These have been used to make comparisons with other 32 bit architectures. As a commercial product details of its derivation have not been published, although a summary may be found in [Richards82]. This account of its development is derived from conversations with the designer during 1984. CINTCODE has been implemented on a number of machines, and a measure of its success may be seen in the implementation of a multi-tasking operating system, Tripos [Richards79], on the BBC microcomputer, a 6502 based computer with 32Kbytes of memory [Baldwin84].

A CINTCODE instruction normally consists of a one-byte opcode followed by a number of one or two byte arguments. The sole exception is the SWITCH instruction, used to implement the BCPL SWITCHON statement, which has an arbitrary number of arguments, the number of such arguments being given in the first argument byte. The instructions consist of a small core that provide the basic operations to support the high level language, and a substantial number of variants on these core instructions to improve the performance of the instruction set by dealing

[3] For clarity of notation the term *opcode* will be taken to refer to the initial opcode byte, and the term *instruction* will refer to the opcode with its arguments as a whole entity. This is a source of considerable confusion throughout the literature, with the two words often used interchangeably.

with special cases efficiently. The machine has three general registers, that essentially hold the top three elements of the local stack frame (A, B and C), and three registers to provide access to local data (the stack pointer P), global data (the global base register G) and the code (the program counter PC). There are three basic groups of instructions, which take 2-byte arguments where necessary:

(1) Data access instructions to load and store data from the global and local areas, from an absolute address, and to load constants

(2) Data manipulation instructions that operate on A, B and C.

(3) Flow of control instructions that provide conditional flow of control.

CINTCODE uses three variants on these basic instructions, to provide new instructions that will take up less room.

(1) Variants are provided with one byte arguments, in cases where the arguments are commonly in the range 0 - 255 (e.g. Load Local Byte). A version of this is also provided for arguments in other ranges, e.g. for globals in the range 256 - 511.

(2) Variants are provided for cases where a particular argument is common, in which the argument is implied in the instruction. Thus to load constant 0 into register A we have Load Immediate Zero.

(3) Commonly used sequences of instructions are combined. Thus we have Call Global to load a global value and then do a procedure call to that address.

These new instructions may themselves be combined to provide further instructions, for example Add Constant One.

The choice of which instructions to choose from all the potential variants was based largely on the designer's (considerable) experience of writing BCPL compilers, assisted by a limited amount of statistical data collected by the compiler in everyday use. He felt that about half of the instructions he generated were loading and storing values, in the ratio two load to one store, and this is the density found in the instruction set. The idea of weighting the support given to particular types of operation in the instruction set is the crucial advance in this style of computer design, and must ultimately be attributed to Knuth's work on analysing FORTRAN code. Prior to

this, computers were designed without any consideration of the degree of support for particular criteria. Although CINTCODE was not the first instruction set to make this distinction (a claim that might more reasonably be made by designers of the Alto), it is typical of such designs. Whether weighting the number of instructions of a particular type in proportion to the frequency of the operation in compiled code is correct is discussed in a chapter five below.

A full analysis of CINTCODE used in practice has not been carried out, although the designer estimates a three-fold improvement over existing instruction sets in static size. In chapter two, an analysis of a 32-bit version of CINTCODE in comparison with other 32-bit machines is presented, and this suggests that a two-fold improvement is more realistic.

Designs such as the Alto and CINTCODE all rely in essence on the designers experience of the use of their language in compiled form. Following on from Knuth's work a number of people have looked at the use of statistics to help explicitly in the design of computer instruction sets. The key ideas here come from Tanenbaum, who designed a machine, EM-1 to support a simple block structured programming language, SAL [Tanenbaum78]. In this respect it closely parallels the CINTCODE machine for BCPL. The work is roughly contemporary with the development of CINTCODE, although the two designs were independent. However Tanenbaum made a thorough study of high level language usage, within the environment in which he was interested (small student programs), before designing his machine. It is interesting that many of his conclusions closely parallel those of earlier workers, who had used only their own intuition, and his instruction set is very close to that for CINTCODE.

SAL is a language not dissimilar to BCPL, the one significant extension being the addition of static scope for local variables. The programs under examination were written in a highly structured style, with very short procedures (average 18.2 executable statements per procedure). Tanenbaum collected a large number of static and dynamic statistics about high level constructs. These showed like Knuth that programs are generally very simple. 93% of executable statements are assignment, call or if statements. Expressions are generally simple (usually a constant or a variable), most operands are constants or simple variables. Most procedures have very few formal parameters, and very few scalar local variables. Tanenbaum

suggests that design effort should go into instructions that handle these common simple cases.

EM-1 is a segmented machine with a byte stream instruction set, with separate program and data segments being byte and "word" addressed respectively. In this respect it is unusual, moving away from the standard von Neumann model of computing. The size of a word is not defined, but is effectively constrained to be 16 bits. There is a special data segment for the stack, addressed by a stack pointer register. EM-1 opcodes may have a single or double byte argument. To increase the number of instructions (which normally would be 256 for a byte stream instruction set) one opcode is reserved as an "escape" opcode, giving a further 255 instructions. Tanenbaum, drawing on the work of Huffman [Huffman52] and Shannon [Shannon48] points out that the commonest instructions should be given the shortest encoding. In this case with one escape opcode only, it is a rather trivial case of base 256 Huffman encoding.

Tanenbaum's instruction set consists of a number of instructions to support the operations required by SAL forming a "core" instruction set. He then augments his instruction set with additional instructions in the manner of CINTCODE. In fact Tanenbaum attacks the problem from the opposite end to CINTCODE, with his core instruction set containing instructions with single byte arguments. He adds opcodes with double byte arguments to deal with pathological cases, which in practice due to their rarity are implemented as escaped opcodes. This is derived from the assumption that since programs are small, argument values such as branch addresses and stack frame offsets will be small and fit in one byte for the normal program. Further instructions with implied arguments for particularly common cases are then added to complete the instruction set.

The statistics on program usage are used to guide the augmentation process. Thus there are three additional instructions for pushing constants onto the stack, eight for external variables, and twelve for local variables, reflecting the importance of assignment, and the fact that local variables are most common. Tanenbaum points out that although he is attempting to reduce static size, this will also affect program execution time, by reducing processor-bus bandwidth, and reducing working set sizes. To assist in improving performance he consults his dynamic

statistics, although as he points out these may be dominated by a few very frequent loops, and thus really reflect a much smaller sample of compiled code. With modern MIMD machines, based on processors such as the INMOS Transputer [Cownie86], the desire to keep the amount of memory per processor to a minimum, to save physical space and heating problems makes reduction in static size more important than ever.

Statistics are given for the number of bytes required to compile each high level construct, in comparison with the PDP-11, and a CDC Cyber. These suggest a 3 - 4 fold improvement in size. For complete compiled programs the improvement seems to be more like 2 - 3 fold, reflecting presumably the dominance of particular high level constructs.

As a rider Tanenbaum suggests a solution to the potential nightmare of compiling code for these machines, with the use of a higher level assembly code as code-generator output, and the use of an optimising assembler to select the best instruction in each case. This would also provide more flexibility if the instruction mix had to be changed. This is not so far from the idea of peephole substitution of a core instruction set mentioned in the introduction.

Tanenbaum's work with its use of careful analysis of the target environment represents an advance on designs such as CINTCODE and the initial instruction set for the Alto, which used only minimal statistical data, and were far more dependent on the experience of the machine designer. However he still only used statistical data to guide his design, rather than as a calculus for instruction selection. A more rigorous methodology was followed by Sweet and Sandman in their work to refine the MESA instruction set. Although this was not designed from scratch, their identification of which statistics were important in instruction set design was valuable. Their calculation of expected improvements was the first use of statistical data to estimate results attainable by instruction set refinement.

The MESA architecture was designed at Xerox to support the MESA programming system. MESA is a high level systems programming language, designed to support the development of large information processing applications. Throughout the project the processor architecture, programming language and operating system were designed as a single unit.

The three main goals of the architecture were stated to be [Johnsson82]:

- to enable the efficient implementation of a modular, high level programming language such as MESA. The emphasis here is not on simplicity of the compiler, but on efficiency of the generated object code and on a good match between the semantics of the language and the capabilities of the processor.

- to provide a very compact representation of programs and data so that large, complex systems can run efficiently in machines with relatively small amounts of primary memory.

- to separate the architecture from any particular implementation of the processor, and thus accommodate new implementations whenever it is technically or economically advantageous, without materially affecting either system or application software.

The processors are general, in the sense that they aim to support any high level block structure language, such as Pascal or Algol, and not just MESA, providing instruction set support for those constructs that are common to such languages. The instruction set, like those discussed before, is asymmetric, with the size and number of instructions based on frequency of use. It is a stack machine, and has no general purpose registers. The basic unit of data is the 16 bit word within a 2^{32} word virtual address space in 256 word pages. Asynchronous processing is supported via monitors and condition variables.

The instructions consist of a one byte opcode, followed by a number of byte or word arguments (the format being implied from the opcode). The generic instruction has word sized arguments, with special versions for commonly used cases having byte arguments, or implied arguments. There are also instructions for common sequences of generic instructions. Compilation is by use of the generic instruction, which may then be peephole optimised to one of the special case instructions.

In their paper Sweet and Sandman describe the refinement of the MESA instruction set in the light of statistics on its use [Sweet82]. They provide a formalisation of the statistics required, and the method to be used. Their motivation was to reduce the static size of compiled code, since the Alto (an early 64Kword MESA architecture, without virtual memory) had run out of address space. They were already starting with an instruction set that had been influenced by the designers' experience of compiled code and included support for common instructions in the

manner of CINTCODE and EM-1. They proposed a five stage design process to refine this instruction set.

(1) *Normalise the object code* - Many of the instructions in existing compiled code were special versions of generic instructions dealing with common cases. These were replaced by their generic instructions, with all arguments explicit and word sized. In the case presented by Sweet and Sandman this gave 2.5 million bytes of code to analyse.

(2) *Collect Statistics* - The resultant 2.5million bytes of code were analysed by pattern matching to find particular statistical information.

(3) *Propose New Instructions* - The statistics from stage two were used to suggest new instructions.

(4) *Peephole Optimise* - The object code was converted to use the new instructions.

(5) Repeat stages 2 - 4.

Normalisation is an important concept in refining instruction sets. It is important to get rid of any existing assumptions about beneficial instructions. The existing instruction set utilised 240 out of a possible 256 instructions. The normalised instruction set had 100 generic instructions, giving scope for derivation of 156 new instructions.

Five sets of statistics were found to be useful. These were:

(1) *Static instruction frequency*

(2) *Operand values* - A histogram of argument values for each opcode.

(3) *Instruction successors* - A histogram of successor instructions for each instruction.

(4) *Instruction predecessors* - Ditto for preceding instructions.

(5) *Popular instruction pairs* - The set of all instruction pairs, sorted by frequency.

It is clear that (3), (4) and (5) all report the same information. However (3) and (4) are a more useful representation when considering conditional probabilities. These statistics were used to guide the choice of new instructions. Note that there is no formalisation of the mechanism of choice.

At the end of the process a new instruction set was obtained, with in fact more than 256 new

instructions, so that an escape opcode as in EM-1 was used. Sweet and Sandman were able to use their statistics to predict the saving in space that would accrue from their work, 12%. Their subsequent peephole substitution of the refined instruction set showed this prediction to be accurate. This improvement may not seem dramatic, but bear in mind that they were competing with a machine that had already been designed by some very experienced computer architects. The 12% is a useful figure in indicating that mechanical analysis can have a benefit over human knowledge and experience. Chapter 3 of this dissertation presents a system of mechanised instruction set design that yielded 14% saving compared with CINTCODE in an instruction set for BCPL, and 71% saving in refining an instruction set for the programming language POLY [Matthews85].

1.6. Current Research

Byte stream architectures of the sort described earlier in section 1.5 have had considerable influence on current computer designs. Relatively few large modern computers are based purely on byte stream instruction sets. The Xerox D-machines, descendants of the Alto are perhaps the most important, although a number of 8-bit micros (for example the 6502 and the Z-80) used such designs. Byte stream instruction sets are more common with soft microcoded machines[4] used for experimental work in which the ability to reconfigure the instruction set is a valuable asset. Examples of such machines are the ICL Perq [3rivers81] and the High Level Hardware Orion [HLH85]. However modern mainstream architectures have all been influenced to some extent by the need to support compiled high level languages in the computer instruction set. Some machines such as the VAX-11 [DEC81] and the Motorola MC68000 [Motorola79] have provided generally useful features, such as software stack support and procedure calling. Others attempt to support a specific high level language. The NS32000 series is designed to support Modula 2, with a falling software stack, scaled addressing and procedure call via link tables [National83]. There is also some recognition of the need to support particularly common constructs with additional instructions. Both the MC68000 and the NS32000 have special

[4] That is microcoded machines in which the microcode is dynamically loaded from backing storage.

addressing modes to handle small constants efficiently. These are all conventional von Neumann machines, suited to imperative procedural languages such as FORTRAN, COBOL, Pascal and even to some extent LISP. More radical machines have had some success when dealing with other types of computer language. The Symbolics LISP machine runs LISP far more efficiently than conventional architectures, and has had considerable commercial success. The combinator machine SKIM [Clark80] handles functional languages efficiently, and there has been considerable work on architectures to support logic programming. In all these designs the needs of compiled code have paid a large part in the choice of instruction set.

Recently there has been criticism of the complexity of instruction sets on machines such as the VAX-11. Even the humble Z80 has over 700 different instructions. The VAX-11 with its three address instructions has an even more complex instruction set. There are for example two different instructions for evaluating a polynomial. The RISC's described earlier in this chapter are a reaction to this heavyweight design approach, and have led to much faster and smaller processors.

Figures have been given earlier in this chapter showing the performance of a recent RISC design, the Acorn RISC Machine [Wilson86]. It is worth looking at the first RISC design, RISC-1, to identify the key elements in its design [Patterson82]. The aim was to produce a computer that was effective and simple to implement as a single chip, which led to four constraints.

(1) One instruction are executed per cycle.

(2) All instructions are the same size.

(3) The only access to memory is to load and store registers, all other operations operate on internal registers.

(4) The design is optimised towards high level language support.

RISC-1 meets these requirements. The success of its simple approach is seen in the fact that it needed only 44500 transistors and that the first silicon worked with only one minor design error. The processor supports 32 bit addresses and 8, 16 and 32 bit data. It is optimised to support C. Analysis of compiled C code for the VAX-11, PDP/11 and MC68000 showed that the most time-consuming activity was procedure call, and that most operands were local scalars or constants.

These are the operations concentrated on. There are a total of 31 instructions, each of 32 bits. Each instruction may optionally set a condition code, specify a destination register, and either one or two source registers, or an immediate operand. There are a total of 32 registers, with register zero permanently containing zero. By using r0 the number of memory addressing modes is increased. Procedure call is facilitated by the use of overlapping register windows. RISC-1 has a large bank of registers, and on procedure call allocates a new set. Not all 32 are reallocated however. r0-r9 are global registers and never change. After a call registers r10-r15 become registers r25-r31 of the called routine, and a new set of registers r10-r24 is allocated. This facilitates passing of arguments, as well as reducing memory accesses to save registers. On overflow the registers are dumped to memory by a software trap. To facilitate addressing on the stack the register bank is mapped into the address space. Patterson and Sequin [Patterson80] report work that suggests only 1% of all procedure calls will cause a software trap if there are eight register banks. The data presented suggest that overlapping registers lead to a considerable reduction in memory access due to saving and restoring of registers on procedure call. As a consequence of the constraint on performing one instruction per cycle RISC-1 implements delayed jumps. It is not possible to read an address field from an instruction and carry out the jump in one cycle, so the jump is delayed for one cycle. This is common in microcode, for example the High Level Hardware Orion [HLH85], and is easily handled by a compiler. It does emphasise the fact that RISC-1 is a high level language machine, and not for the assembler programmer.

RISC-1 represents a turning point in computer architecture, but not because of the high performance or novel instruction set. Patterson and Ditzel [Patterson80] were the first to recognise the significance of the *semantic gap* in instruction set design. Their work throughout is based on the assumption that the information in the high level language program must be conveyed to the the low level processor with the introduction of as little redundant information as possible. In giving the compiler writer very simple building blocks he has the opportunity to translate each high level construct without introducing any extra ideas into the compiled code. With the instruction set of the VAX-11 finding an exact semantic match is difficult. The instructions are so complex that even with the large number of available instructions the chance of having precisely the right flavour available is low. Having found a match the chance of the

operands and result being in the right place is also quite low. If the compiler writer does succeed it is often because the instructions are so general they run painfully slowly. On the NS32000 up to 57 cycles can be spent decoding operands for an instruction alone [Wilson86]. By comparison RISC instructions are generally simple enough to execute in one cycle.

RISC technology, like byte stream architecture before has started to influence new computer designs. Many manufacturers are now bringing out RISC designs. Other architectures show RISC influence, for example the Inmos Transputer has kept its instructions simple enough to execute in a single cycle, although overall it is far from being a RISC design [Cownie86].

1.7. Theoretical Studies of Instruction Sets

In parallel with work on actually designing instruction sets, a number of theoretical analyses of the problems involved have been carried out. The whole area is clearly likely to be amenable to information theoretic ideas, and indeed Huffman's paper [Huffman52] is a common reference throughout the literature. Other approaches have also been taken in an attempt to estimate the amount of "improvement" that can be obtained through instruction set design.

Wade and Stigall used information theory, and the concept of entropy to attempt to estimate how much space compiled code for a particular high level language should require [Wade75]. Their approach was to treat the instruction stream as a sequence of symbols and consider the information content of the stream and calculate its entropy. Shannon [Shannon48] defined the information content of a symbol, s_i in a stream of symbols as:

$$\log(\frac{1}{p_i})$$

where p_i is the probability with which s_i occurs. Shannon showed that it is convenient to use logs to base 2,[5] when the information given is measured as bits per symbol, the number of bits that should be used to encode a symbol in an optimal encoding of the instruction stream. The

[5] For the rest of this section logarithms should be assumed to be to base 2, unless explicitly subscripted.

average information per symbol is

$$H = \sum_{i=1}^{N} p_i \log\left(\frac{1}{p_i}\right)$$

$$= -\sum_{i=1}^{N} p_i \log\left(p_i\right) .$$

This quantity is referred to as the entropy of a symbol stream by analogy with similar systems in statistical mechanics. It represents the average bits per symbol that are needed in an optimal encoding of the symbol stream as binary numbers. Huffman [Huffman52] gives an algorithm for assigning an optimal encoding.

Treating the opcode stream as a sequence of such symbols is particularly informative, since then the problem of minimising code size becomes a problem in minimising the entropy of the opcodes. Wade and Stigall considered first the replacement of a given sequence of opcodes by a new opcode wherever it occurred. Suppose the string of opcodes $s_1 s_2 \cdots s_k$ occurs with probability p_s and wherever it occurs is replaced by a new symbol s', then Wade and Stigall showed that the new entropy H' is given by.

$$H' = -\frac{p_s}{1-(k-1)p_s}\log\left(\frac{p_s}{1-(k-1)p_s}\right) - \sum_{i=1}^{k}\frac{p_i - p_s}{1-(k-1)p_s}\log\left(\frac{p_i - p_s}{1-(k-1)p_s}\right) - \sum_{i=k+1}^{N}\frac{p_i}{1-(k-1)p_s}\log\left(\frac{p_i}{1-(k-1)p_s}\right) .$$

Adjusting for the new length of the string of symbols they derived a formula for the reduction in entropy. They then considered the values of p_s for which this value is positive (i.e. it is a reduction, not an increase!). For small p_s he shows that this is given by:

$$p_s > e \prod_{i=1}^{k} p_i .$$

where e is the base of the natural logarithm. Thus there is a decrease in entropy if p_s is greater than

$$2.72 \prod_{i=1}^{k} p_i \ .$$

This quantified the idea that replacing a common sequence of opcodes by a single opcode is beneficial. It also justified the suggestion that rarely occurring opcodes may be replaced by a stream of common opcodes.

Wade and Stigall also considered the idea of introducing state into a processor to modify the action of opcodes. They divided the symbols into K groups, such that there are some symbols which occur in only one group. Each time a group occurs it is preceded by a state symbol, and the symbols unique to the group may share a representation with symbols unique to other groups. Clearly the introduction of extra symbols means that we need to consider the break even point at which it is worth having such symbols. A key factor here is the total probability of all unique symbols over all groups. Clearly the higher this is the greater the benefit of dividing. Wade and Stigall calculated the break even point for various lengths of symbol stream, numbers of groups, and total probability of unique symbols. They also considered the case K=2 and worked out the saving in entropy for different total unique probability and symbol stream lengths. The equations are large and are not repeated here, but they showed that for opcode streams of length 500, 2 groups, and all opcodes in only one group a saving of 0.96 bits per symbol was possible. For smaller streams, larger numbers of groups and opcodes that are not unique to groups the savings are smaller.

Hammerstrom and Davidson used a similar analysis to look at the information content of memory references in the IBM S/360 [Hammerstrom77]. They considered the entropy of references as an n-order Markov process. If $H_n\ (S\ )$ is the entropy of a stream of symbols S analysed as a n-order Markov process, then the absolute entropy is $\lim_{n \to \infty} H_n\ (S\ )$. This is clearly only valid for an infinitely long stream of symbols, and so they derive a formula for an estimate of H, $\hat{H}$. This model was applied to data for the IBM S/360 architecture, using base and displacement addressing. The estimates obtained suggest that the average information content of such an address might be quite low, in one of their two sample programs as little as 0.1 bit per reference. Given an IBM base and displacement address requires 16 bits this represents over 99% redundancy.

As a theoretical basis for instruction set design entropy shows great promise. Although a detailed study is beyond the scope of this dissertation, some analysis of the entropy of byte stream instruction sets is discussed in chapter five.

Flynn and his co-workers took a rather more pragmatic approach and derived a *Canonical Interpretive Form* as a best possible instruction set for a particular program, suggesting that for FORTRAN, compiled code 10% of the size of the best current compiled code should be possible [Flynn84]. Flynn restricts his analysis to a single high level language, in which all optimisations have been done at source level. He defines the concept of an ideal execution architecture, where by ideal he means:

(1) *Transparent* - i.e., translation is a simple process which preserves equivalent source-state information, thus allowing a ready reconstruction of source constructs.

(2) *Optimal representation* - i.e., space and time to compile and execute are minimised.

Translation (i.e. compilation) is viewed as providing a reduction in computational complexity, through the partial binding of operands to addresses and operators to computational structures. Remember that any optimisation has already been done, on a source to source basis.

Flynn goes into some detail on transparency. In general there may be any number of levels of interpretation, from the high level language to the cpu electronics. Transparency between two levels implies that state transitions in each level occur in the same order, and that in the transition from state to state there is a one to one correspondence in states, and no new states are introduced.

To clarify his views on optimising space and time Flynn introduces the concept of an *environment,* that is all the information (registers, memory, interrupt state etc.) needed to interpret a particular instruction. Within this concept the idea of *distance* is introduced. A program will typically enter some environments repeatedly during execution. *Distance* is the number of distinct environments entered. *Distance ratio* is the number of distinct environments

entered as a fraction of the total environments entered. The smaller this value, the greater the opportunity for re-using state.

With these concepts Flynn defines five criteria for measuring architectures:

(1) *Correspondence* - We want the low level representation to correspond to the source representation. Correspondence may be viewed at many levels (program, procedure etc.), but from the point of view of transparency, the high level language operation is the useful unit of distinction. For any source program we define

$$\hat{A}_i \quad A_i \quad \hat{A}_d \quad A_d$$

to be the number of static operation references, dynamic operation references, static data references and dynamic data references respectively.

$$\hat{a}_i \quad a_i \quad \hat{a}_d \quad a_d$$

are the corresponding counts for an actual execution architecture. The quadruple:

$$\frac{\hat{A}_i}{\hat{a}_i} \quad \frac{A_i}{a_i} \quad \frac{\hat{A}_d}{\hat{a}_d} \quad \frac{A_d}{a_d}$$

is used as a measure of correspondence.

(2) *Size* - We want a minimal encoding. This is merely the product of the dynamic counts above, and the size indicated by the number of distinct operators (F) and operands (V)

$$A_i \left\lceil \log_2 F \right\rceil \quad A_d \left\lceil \log_2 V \right\rceil$$

There is a slight problem here, since in a general programming language the number of data operands may be arbitrarily large, and for practical purposes we wish to restrict the value, so that the size of identifiers is finite. Flynn chooses the program as the correct environment over which to carry out the analysis. This should perhaps be the largest program we ever wish to run. An alternative would be the subroutine, since in many programming languages there is locality of scope within subroutines. Flynn also (deliberately) ignores saving possible through frequency based encodings.

(3) *Referencing Activity* - this is composed of both instruction fetches and data stores and fetches. For instructions the number of references for the operation and operands is not simply $A_i + A_d$, since A_d will be two for operands used both as domain and range. Instead Flynn uses the concept of the dynamic syllable count:

$$A_s = A_i + \sum_i (\hat{A}_d)_i$$

The data reference count is given simply by A_d .

(4) *Stability* - this is a measure of the number of environments encountered, the number of computed locations and the number of potential disruptions to serial instruction sequencing. It is measured by the triple

$$S_e \quad S_c \quad S_b$$

where S_e is the total number of environments encountered during executions, S_c is the total number of data references that have to be computed (e.g. record elements) and S_b is the number of control actions dynamically interpreted.

(5) *Distance* - This is a measure of how often we have first encounters (i.e. first reference to a branch address, data operand etc). It is given by the triple

$$D_e \quad D_c \quad D_b$$

where D_e is the number of unique environments entered, D_c is the number of computed addresses and D_b is the number of unique branch targets.

With these definitions Flynn comes up with his five canonical interpretive measures of an ideal execution architecture.

(1) *Correspondence* - measured by the quadruple $\hat{A}_i$, A_i, $\hat{A}_d$, A_d.

(2) *Size* - Each operator is of size $\lceil \log_2 F \rceil$ bits, each operand of size $\lceil \log_2 V \rceil$ bits.

(3) *Referencing* - Syllable references are measured by A_s.
- Instruction references are measured by the pair M, A_i.
- Data references are measured by A_d.

(4) *Stability* - measured by the triple S_e, S_c, S_b.

(5) *Distance* - measured by the triple D_e, D_c, D_b.

Using these measures Flynn is able to calculate ideal execution space, i.e. the best possible static size of a program as

$$\sum_{\alpha} \left\lceil \log_2 F_\alpha \right\rceil \times \hat{A}_{i\alpha} + \left\lceil \log_2 V_\alpha \right\rceil \times \hat{A}_{d\alpha}$$

where the summation is over all environments α.

Flynn calculates ideal execution time as

$$aA_i + bA_d + cS_e$$

where a, b and c are arbitrary relative weights for transformational activity, data accessing and environment change. The effect of a cache is to reduce a and b substantially, so that environment change may dominate. *Canonical Interpretive Form* is defined as being an interpretive form (that is instruction set) which has ideal execution space and time.

The presence of the weighting factors a, b and c in the estimation of ideal execution time, rather limit the use of this measure, but Flynn does give some comparisons between ideal and actual execution space. For the three line FORTRAN fragment

```
I = I + 1
J = (J - 1) * I
K = (J - 1) * (K - I)
```

the ideal execution space is 30 bits (6 operators and 9 operands, each requiring 2 bits). This compares with FORTRAN-H compiled code for the IBM S/370, which is 368 bits, over 12 times the canonic measure. For a rather more complex Pascal program the CIF measure of execution space is 277 bits, compared with 2800 bits for the PDP 11, 3056 bits for the IBM S/370 and 4960 bits for UCSD P-code.

Flynn's measures are based on somewhat simplistic views of the world, and again assume that

bit-addressing is the norm. In addition he does not consider the space required to describe the encoding he is using (you may only have 5 unique branch addresses, but at some stage they have to be identified). He has tried to apply his ideas, and come up with instruction sets, *Adept* to support Pascal, and *DELtran* to support FORTRAN, but in the Pascal example above Adept still requires 1184 bits, four times the CIF ideal execution space.

These theoretical studies are very much of one type; they give a lower bound beyond which it is not possible to improve. As some of Flynn's work shows such results have little to do with reality. A far more useful result would be an upper bound rather than lower bound on improvements possible in code size and speed. In chapter five we consider an approach to achieving just such an upper bound.

1.8. A New Methodology

In demonstrating the importance of the semantic gap in instruction set design, RISC technology has provided a model which shows great promise. The RISC approach suffers however from its very verbose encoding and has lost the conciseness of representation found in byte stream designs. This dissertation will present the thesis that it is possible to design an instruction set that bridges the semantic gap between hardware and software as successfully as RISC and yet has the conciseness of a byte-stream instruction set. The methodology is suited to automation and is generalised to permit refinement of existing instruction sets and to permit a design to meet criteria other than static or dynamic compactness.

2. Experimental Observation

At the heart of work on instruction set design lies careful study of the environment in which the instruction set is to work. For experimental purposes it is not necessary (nor even desirable) that a full blown computer system and environment be used, but it is wise to select a subject for analysis which is big enough to provoke the same problems. My experimental subject is the Tripos operating system, and in particular the running of commands under the Tripos command line interpreter.

Tripos [Richards79] is a small real-time operating system written in BCPL [Richards80]. It implements message passing by shared memory for interprocess communication. There is no support for virtual memory, nor is there any form of memory protection. In the system under analysis it runs as a single user operating system on a Motorola MC68000 [Motorola79] processor. The sole interface with the outside world is through a local area network, a Cambridge Ring. The network interface is managed by a separate 6809 based I/O processor. All services are provided remotely through the local area network. This system is well described in [Needham82].

The Tripos system supports a simple command line interpreter as interface to the user with commands in general written in BCPL. It is useful to look at the use of commands both in relation to other commands and in relation to the overall activity of the processor. This is described in the first half of the chapter and used to choose an appropriate design criterion for a BCPL instruction set to support commands running within a distributed Tripos environment. The second half of the chapter is devoted to experimental analysis of BCPL as used to write Tripos commands. Those aspects particularly relevant to the chosen design criterion are examined in detail. These results are used in the next chapter to guide the design of an instruction set to support BCPL.

2.1. Command Usage

By virtue of the machines under analysis being linked by a local area network it is possible to analyse their everyday use remotely. One simple preliminary analysis is to look at how much of a machine's time is spent carrying out various tasks (for example command execution, file handling, terminal I/O). A sampling technique was used to look at all the relevant machines in active use on the network. The network resource manager was interrogated to find which machines were in use, and each machine was interrogated in turn to discover which process was active at the time. The interface from the host MC68000 processors to the network has its own 6809 processor, which is capable of reading the host machine's memory via a DMA link. This was used to investigate system data structures to find the currently active process. Using the interface processor leads to minimal influence on the performance of the host machine, an important factor in this sort of analysis. The sampler repeatedly analysed each active machine, building up a profile on the frequency with which different processes were found executing. The sampling machine was masked out of the analysis to avoid biasing the results. Table 2.1 summarises the results, rounded to the nearest 1%. These data represent a total of 31579 samples, taken during a typical busy weekday afternoon. A total of 11 processors were analysed, although not all were necessarily active the whole time. These are average results (weighted by sample size) for all the machines, together with the maximum and minimum figures for individual machines, to give a feel for the spread of activities. The time for the command line interpreter process represents that percentage of time dedicated to running user programs. We

Process Details	Percentage Time		
	Average	Minimum	Maximum
idle process	84%	60%	97%
command line interpreter	11%	2%	13%
terminal handler	2%	0%	3%
file handler	1%	0%	1%
all other processes	2%	0%	10%

Table 2.1 Processor Usage

might reasonably assume that some part of the time in the idle process is due to logged on users not actually doing anything. However it is striking that 60% of the time is spent in the idle process even in processors which were running chip simulation programs continuously throughout the period. In other words within this environment there is inevitable idle time, presumably due to communication delays on fully utilised processors. If we can reduce this we can improve system performance.

It is interesting to look at what sort of programs are executed. The sampling program was modified to identify the command being executed by the command line interpreter. As a result of 12575 samples, the data in table 2.2 were obtained. The figures are as a percentage of total time in the command line interpreter. It can be seen that users spend most (84%) of their time sitting in large interactive systems, with occasional demand on batch processing programs. In particular the large number of small utilities (23 out of 30 programs analysed) take very little time (2%). It is apparent that there is little to be gained within this environment by aiming to increase overall processor throughput, since it would appear that the user is not using the processor to full capacity anyway. However there would seem to be some benefit in increasing processor throughput in short bursts, to improve response when the user runs a program.

Execution of a command by the user involves two stages, the loading of the command, and then the execution of its code. It is worth looking at which of these dominates, and thus which has

Program Type	Percentage Time
editors	53%
interactive LISP	31%
paginator	10%
BCPL compilers	2%
file transfer (MVSCP)	2%
system utilities	2%

Table 2.2 Types of Commands Executed

greatest scope for improvement.

The command line interpreter was modified to log details of each command that was executed, together with load and execution times to a remote machine. This modified program was installed in the standard system to monitor typical usage. Note that the logging was done after command execution was complete, using a very simple communication protocol to minimise interference with system behaviour. Table 2.3 summarises the results. About the time these results were being obtained a system for preloading common commands was being introduced, reducing to zero the load time for such commands. Such results are separated and shown in the first line of the table. We see that the majority of commands executed take very little time to execute, but spend a relatively large amount of time being loaded. In total the time may not be very large, but the effect of removing the burden of loading, as is achieved by the system of preloading commands, is to give improved response from the computer. It is perhaps worth looking in more detail at the nature of loading commands, to see if there is scope for improvement in this area.

One way of improving response, as already hinted above, would be to preload commonly used commands in main store. To get a feel for how much of the load time is due to waiting for the secondary storage device (as opposed to the housekeeping of handling files), the standard remote disc filing system was compared with an in-store "silicon disc" filing system. Rather than

Command Type	Count	Load Time Average/s	Execution Time Average/s	Load Time as Percentage of Total
Preloaded	126	0	80.8	0%
Under 1s execution time	175	1.12	0.22	83%
1s - 10s execution time	99	2.42	3.62	40%
10s - 100s execution time	44	3.12	28.1	10%
Over 100s execution time	15	5.85	880	0%
Total non-preloaded	333	2.00	44.5	4%

Table 2.3 Command Load and Execution Times

rely on random observation, a simulation of the basic action of loading a command was used. A program was written repeatedly to open a file, read each byte of it and close it. The remote sampling program described at the start of this section was then used to compare two processors, one running the test program reading a file from the file server, the other with a file on the silicon disc. There are three processes of interest; the command line processor running the test program, issuing requests to open read and close files, the file handler process, which either obtains data from the silicon disc image or issues requests to the remote file server for data, and the idle process, which is entered when neither of the other two processes may proceed. The program that uses the remote file clearly will spend some time waiting for the file server to respond to requests, and this time will be spent in the idle process. By looking at this idle time we can get an estimate of the amount of time spent reading files that is due to waiting for file server response. This is an upper bound on the time that could be saved by preloading files. However use of a silicon disc means the local processor has to carry out additional work with file handling that was previously done by the remote file server. By looking at the ratio between execution time spent in the command line interpreter and in the file handler process we can get a measure of how much extra work is being done by the local processor in running the silicon disc. This will give us a lower bound on the amount of time that would be saved through using a silicon disc. Table 2.4 shows the first of these results, the time spent idle when reading

remote files. The experiment was carried out three times using files of size 0, 1024 and 131072 bytes, and results are based on about 1125 samples in each case. The figures for 0 bytes and 1024 bytes are the most interesting, since they straddle the size of most small executable commands. We see that there is a very large overhead in handling these files, with at most 22% of the processor time actually used. The larger figures for small files are because of the overhead in opening a file and initiating transfer protocols. If we could eliminate the overhead of remote file reading when loading small commands we might expect between a five fold and twenty fold decrease in load times. Even for large files a halving of load time seems feasible.

Table 2.5 shows what happens to the time actually used in processing. For the system under consideration this is divided between the file system handler process and the command line process requesting the read operation. In using a silicon disc the overhead of having a local file system should increase the proportion of the time spent in the file handler, and this can be used to revise the estimate of the improvement possible in loading times. Table 2.5 shows the ratio of file handler to command line interpreter times for remote file and silicon disc for each of the three file sizes used above. The results have not turned out as expected. The silicon disc handles a read request using between one tenth and one half of the processor cycles required by the remote file handler. Although the filing system is local, the administrative overheads it represents are less significant than the time in handling communication protocols to the remote file server. This is perhaps emphasised by the fact that there is a bigger difference for smaller files, for which the overheads of initiating a transfer represent a greater percentage of the total activity. The expected decrease in loading times by use of a local file system could be considerably more than twenty fold.

File Size	Percentage Idle Time
0 bytes	95%
1024 bytes	78%
131072 bytes	53%

Table 2.4 Idle time reading remote files

File Size and Type	Ratio Filing to Reading	
	Remote File	Silicon Disc
0 bytes	~4	0.82
1024 bytes	2.0	0.19
131072 bytes	0.35	0.15

Table 2.5 Efficiency of Filing Systems

To minimise the effects of uncontrollable environmental factors, such as load on the local network these experiments were carried out on identical machines in parallel at a weekend. The results must be regarded as biased in favour of the remote file server. Under weekday conditions of heavy network traffic and demand on the file server there is some evidence that up to 98% of file loading time is spent waiting for the file server.

These experiments taken together indicate that within the environment being studied there is considerable benefit to be gained from improving load times, rather than execution times of commands. Since these experiments were carried out the preload filing system mentioned earlier has been fully introduced into the Tripos operating system [Wilson85]. This is more efficient than the silicon disc in that commands are preloaded ready for execution, and a request for loading of a preloaded file results in a pointer to the file image in core being handed back, with no copying in store. Multiple requests for the same program by several processes results in reentrant use of the single program image. This system of preloading has resulted in substantial improvements in response and reduction in load on the file server. It has also led to an increase in the size of main store needed by users. Whereas 500K was adequate for most purposes before preloading was introduced, now 1Mb or 1.5Mb is preferred, in order that sufficient commands may be preloaded.

This last statement underlines what must be an important aim in providing an instruction set to support a Tripos command environment - reduction in static size of compiled code. This will lead to a reduction in space requirements for preloaded programs, making machines with small

amounts of memory suitable for use with such a system, and allowing better use of the memory on large systems. By increasing the use of preloading the demand on the remote file server becomes less, so improving its performance when it is used. Filing caches, both in the local machine and the file server will work more efficiently with smaller programs. Instruction caches will have a higher hit ratio. Under a virtual memory environment (which Tripos does not possess) working sets would be reduced, a point discussed by Tafvelin and Wikstrom [Tafvelin75]. In the remainder of this chapter we look at the BCPL language, used for almost all Tripos commands. Considerable emphasis is placed on obtaining statistics about the static size of compiled BCPL. This information will be used in chapter three in designing an instruction set for BCPL with the design criterion of minimising the size of compiled code.

2.2. Language Usage

In any design work it is important to get a feel for the overall structure of the language being supported. Many of the measurements described in this section are similar to those carried out by others, as described in Weicker [Weicker84]. Tanenbaum's analysis of SAL is particularly relevant because of the great similarity between BCPL and SAL [Tanenbaum78]. It should be emphasised that the aim of the current analysis of BCPL is not to provide a comprehensive statistical analysis of every facet of the language as such statistics are not of direct use for the design methodology proposed in the next chapter. The analysis is far more about giving the designer a feel for the high level constructs that make up the language and how they are used.

High level analysis of the structure of BCPL programs was carried out by modification of the SYN and TRN stages of the BCPL compiler. The SYN stage takes BCPL source and generates a tree representation of it, the AE-tree. The TRN stage takes the AE-tree representation of the BCPL source, and from it generates OCODE, an intermediate code for code generation. Analyses were carried out on 102 commands from the Tripos system, representing 1020 BCPL procedures, and an average of 19.6 statements per procedure *(c.f.* 18.4 statements per procedure for SAL).

The first experiment looked at the static frequency of the various BCPL statement types. Table

2.6 shows the results for each BCPL statement as an absolute figure and as a percentage of the total. The LET statement is included, since although it is a declaration it also implies an assignment. There is no attempt to weight results according to complexity of the command, these frequencies are straight counts of the number of occurrences of each construct in compiled code. In line with other researchers' results these figures are dominated by assignment, procedure call and conditionals (82.7%). The precepts of structured programming have clearly not yet totally got home to the programmers who produced this sample. GOTO is still far too popular! A more concise representation of this data is given in table 2.7, which groups statements into different types. For comparison Tanenbaum's results with SAL are given. The similarity between the two sets of results are striking. There is no obvious explanation for the rather smaller number of assignments in BCPL. The discrepancy in the use of "unstructured" flow of control statements is perhaps surprising. This is in part due to the relatively large number of flow of control statements possible in BCPL (GOTO, RETURN, LOOP, BREAK, ENDCASE and RESULTIS). In addition RESULTIS is really part of an expression, rather like a function and perhaps should not be counted as a statement. There is no equivalent of the VALOF block in SAL, and if RESULTIS were excluded BCPL would have only 7.5% of statements under the heading of flow of control. Finally SAL is used to teach students structured programming, and so there is presumably rather more pressure to use "structured" techniques than is found in the research environment from which the BCPL sample was taken.

Looking at high level constructs is not always the most helpful way of analysing language use, since some statements are inherently more complex than others, and with almost any instruction set would take up more space. A FOR statement is inevitably more verbose than a simple assignment. An assignment of a constant to a simple variable is inevitably less verbose than assignment of the result of a function call to an element of a vector. To get a feel of the relative size of constructs it is helpful to look at compiled code. This immediately causes problems, because of the presuppositions built into any code used for compilation. A number of remedies are possible. We could collect data for a wide variety of target machines. However data would then be biased in the general direction of current architectural trends. We could collect data on an "objective" sample of target instruction sets. Even if such a sample exists we would however be immediately prejudging the issue. We could use an iterative process, repeating the analysis

Statement	Absolute Frequency	Percentage Frequency
assignment	5503	27.5%
procedure call	5215	26.0%
LET declarations	2459	12.3%
IF	2084	10.4%
RESULTIS	1010	5.0%
ENDCASE	679	3.4%
UNLESS	617	3.1%
TEST	544	2.7%
FOR	481	1.4%
GOTO	266	1.3%
RETURN	226	1.1%
BREAK	181	0.9%
SWITCHON	167	0.8%
UNTIL	166	0.8%
LOOP	139	0.7%
WHILE	109	0.5%
REPEAT	105	0.5%
REPEATUNTIL	47	0.2%
REPEATWHILE	32	0.2%
FINISH	12	0.1%
Total	21907	100.0%

Table 2.6 Frequency of BCPL statements

Statement Type	Frequency	
	BCPL	SAL
assignment	39.7%	46.5%
procedure call	26.0%	24.6%
conditional	17.0%	17.5%
flow of control	12.5%	5.6%
loop	4.7%	5.5%
other	———	0.3%
Total	100.0%	100.0%

Table 2.7 Frequency of BCPL and SAL statements by type

on instruction sets we have designed, but there is no reason to believe this approach would converge on the best instruction set, and it would be an extremely labour intensive approach. The best approach might be to use the size of the translation tree for the construct as a measure. This is perhaps the most abstract representation we can find. The slightly easier road of using the intermediate code, OCODE, generated by the BCPL compiler is perhaps a practical compromise. This already incorporates some assumptions about the world (for example that it has three registers), but is probably closer to the spirit of the high level language than other candidates. Even so the quantisation problems because of the small number of instructions generated for a single high level construct make this a technique of only limited use, and better suited to overall comparison of different instruction sets.

Larger, more heterogeneous structures are better candidates for examination by this technique, with fewer quantisation problems. I have looked at the size of blocks generated by various loop and conditional statements. The size of these blocks is an estimate of the size of offset needed to reference one end of the block from the other. The average size of OCODE blocks generated by some typical statements is shown in table 2.8. It can be seen that the average size of a block is quite small, and relative addressing would certainly seem to be justified by this data. These data are in line with Tanenbaum's observation that the majority (94%) of IF statement

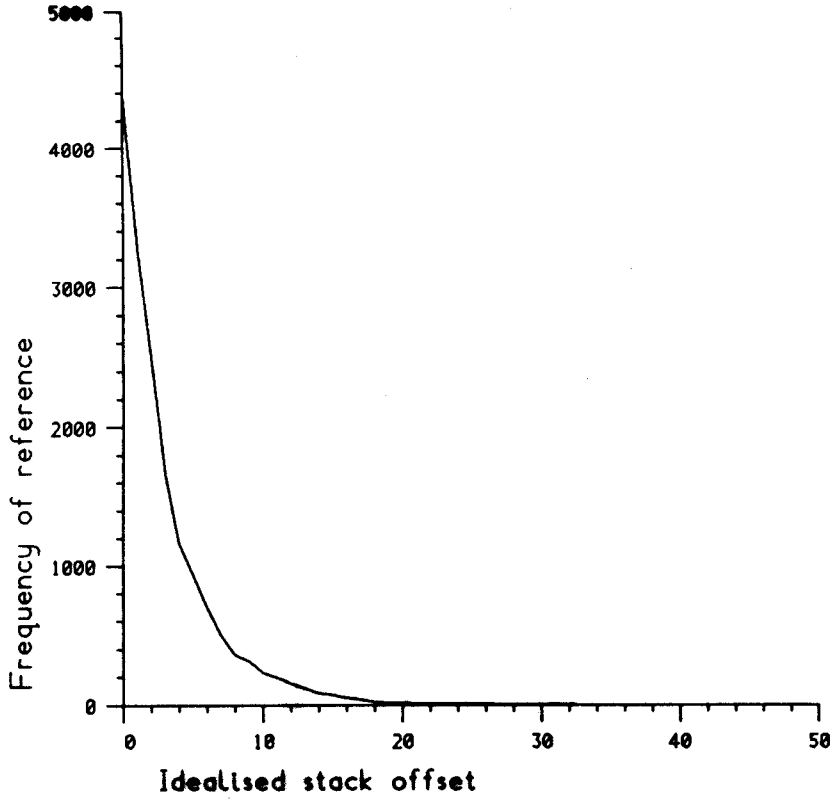
Statement	Average size of OCODE block (bytes)
IF	84
UNLESS	122
TEST-THEN	124
TEST-ELSE	179
FOR	145
UNTIL	194
WHILE	272
Weighted average	120

Table 2.8 Block sizes in BCPL

THEN parts contain less than ten statements. Note that the sign of the offset (i.e. whether we are jumping backwards or forwards) is not relevant in the case of BCPL. We always know at compile time whether the jump is forwards or backwards, and do not have to code this information. It is thus not important that the examples in table 2.8 all involve forward jumps.

Many instruction sets provide optimised handling of the first few variables on the local stack (CINTCODE and EM-1 for example), and it is worth considering whether this is justified. For it to be worthwhile requires a compiler which assigns the smallest stack offsets to the commonest local variables. The BCPL compiler under examination is designed for the MC68000, which gives no significant advantage to such allocation of stack offsets, and such a scheme is not implemented[6]. To enable data to be collected the TRN phase of the compiler was modified to report statistics on local variable reference, which allowed calculation of "idealised stack

[6] It is debatable whether altering the position of local variables is always permissible in BCPL. Certainly it is not permissible to transform the order of procedure arguments on the stack. Whilst it would seem that the proposed BCPL standard allows other variables to be moved, there is some evidence of programmers assuming that consecutive local variable declarations occupy consecutive stack locations. Such "hidden standards" are another trap for the unwary computer designer!



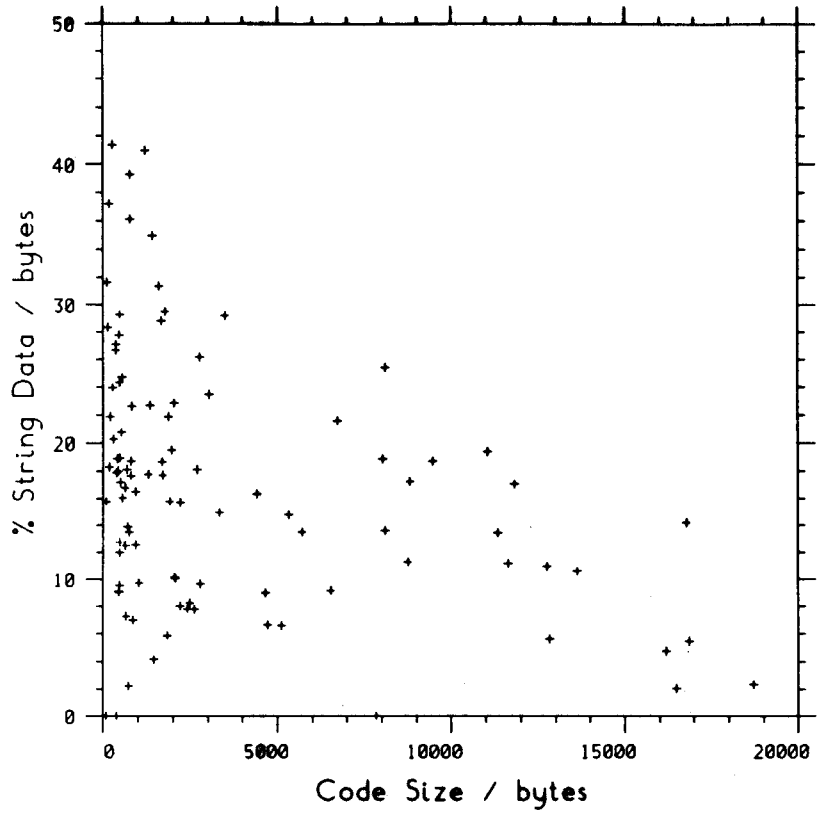
Graph 2.1 Access to local variables

offsets". These are the offsets which would give the most frequent variables the smallest stack offset. Graph 2.1 illustrates the static frequency with which local variables are accessed in compiled code as a function of idealised stack offset. It is clear that specialist support for the first few is justified. These results are very much in line with similar results obtained by Sweet and Sandman with MESA [Sweet82].

Of course compiled BCPL is not only code, but also static data, of which the most significant part is literal strings. Before getting too carried away with the reductions possible in space required by BCPL it is sensible to look at what fraction of compiled BCPL is in fact static data, since this provides a upper bound on the reduction in size that can be obtained from improving the code. For the 102 programs under analysis, which generated 353786 bytes of compiled

code, 44296 bytes, 12.5% of the total, were static data, and this figure provides an upper bound on the reduction in static code size possible through instruction set design. However the amount of static string data in a program is not directly proportional to its size, as is indicated by the scatter diagram in graph 2.2. This shows that smaller programs tend to have a greater proportion of static data, indicating that the potential benefits from denser code are greater for larger programs. This is rather disappointing in the light of our earlier conclusion that small programs (strictly speaking those that do not take long to execute) show most scope for savings in load time by preloading. The amount of static data is very dependent on the type of program. For other systems examined as little as 1% of code was static data.

As was discussed earlier in this dissertation, compiler design can perhaps be as important as



Graph 2.2 Static String Data in Compiled Code

instruction set design in improving efficiency. The effect of different instruction sets on static code size can be seen from table 2.9, which compares three 32-bit instruction sets. The results are referenced to OCODE as a norm. The results for CINTCODE and the IBM S/370 were based on an earlier analysis of code generation from OCODE (not presented here), which gave figures for the ratio of code size to OCODE size, and are rounded to the nearest 1000 bytes. We see a ratio of nearly 2 to 1 between the size of compiled CINTCODE and compiled IBM S/370 code. A 32-bit version of CINTCODE was designed, drawing on some of the ideas of Padget and ffitch [Padget83], to enable a fair comparison. As a result the compiler is perhaps not as well shaken down as the others, and the CINTCODE figure could well be improved given some effort with the compiler.

For comparison table 2.10 shows the effect of using different compilers for the same instruction set, the MC68000. BCPL is the standard compiler and was implemented as a general purpose workhorse to implement Tripos. BCP is a recent optimising compiler, which also aims to provide fast compilation. TBCPL is a commercial product aimed at generating very fast code. This is not exactly ideal for our purposes, since we wish to look at static compactness, as was done with the three different instruction sets in table 2.9. Static compactness and high speed code rarely go together. The amount of OCODE generated is perhaps a useful indication, since it is in the nature of BCPL compilers that most optimisation for speed occurs after the OCODE has been generated. As compilers become more sophisticated the amount of intermediate code generated tends to become smaller because global optimisation techniques remove redundancies. This is

Instruction Set	Code Size (bytes)	Size Relative to OCODE
OCODE	302273	100%
CINTCODE	240000	79%
MC 68000	385936	128%
IBM S/370	465000	154%

Table 2.9 Compiled code size for BCPL using different instruction sets

then countered at the code generation stage by use of in line substitution and loop flattening. We see that for the two compilers where figures are available there is a 2 to 1 ratio in the amount of OCODE generated, even though the amount of compiled MC68000 code finally generated is more or less the same. There is reason to believe that the amount of OCODE generated by TBCPL is even less than for BCP [Evans86], although without the source of the compiler this is hard to verify. The figures show both the wide range of results that can be obtained both by altering the instruction set (nearly a 2:1 factor between IBM S/370 and CINTCODE) and the compiler, and by altering the compiler (nearly a 2:1 factor between BCPL and BCP, and more for TBCPL OCODE). The results for the different compilers on the same machine are perhaps not totally convincing. However there is evidence that the BCP compiler produces code that on average runs twice as fast as that from the BCPL compiler on the MC68000 [Brooks83]. Other examples of variation in compiler performance are also known. The FORTRAN H compiler for the IBM S/370 produces code that runs around twice as fast as that from the FORTRAN G compiler [Bennett81]. It would not be unreasonable to suppose that in the realm of static code size such results were possible. Work by Marks on compilation techniques to improve static code size would certainly support this [Marks80]. It is clear that improvement in computer instruction sets is only one part of improving computer performance. The effect of compiler technology cannot be ignored.

The results presented in this chapter present some insight into the BCPL system we have chosen to investigate. Minimisation of static size of compiled code has been shown to be a worthwhile design criterion. Various aspects of compiled BCPL relevant to its compactness have been analysed. In the next chapter this information is used to design a BCPL instruction set to

Compiler	OCODE size (bytes)	MC 68000 code size (bytes)
BCPL	302273	385936
BCP	153714	395487
TBCPL	———————	380800

Table 2.10 Code sizes for different MC 68000 compilers

meet this design criterion.

3. A BCPL Instruction Set

In this chapter I consider the design of an instruction set to support BCPL as used in the Tripos command environment. This work has led to an instruction set whose compiled code is considerably more compact than that generated for existing instruction sets. The design methods that evolved are brought together into a general design methodology at the end of this chapter.

3.1. The BCPL World View

Any designer must first understand the language he is to support before commencing on design work. BCPL has a very simple structure, but the same basic ideas underlie most imperative programming languages. By its very simplicity it offers the possibility of an approach uncluttered by excessive detail. It is certainly relevant to contemporary design, since although not that widely used itself it is closely related to C.

There are three aspects of the BCPL world; data access, data manipulation and flow of control.

Schulthess in his paper on reduced instruction sets for high level languages comes to more or less the same conclusion for block structured languages in general, although suggesting a fourth set of constructs for stack administration [Schulthess84]. This fourth set seems rather superfluous, particularly in the BCPL world, where dynamic free variables are not permitted. All the operations in the fourth set, except possibly those involved with handling static chains could equally well be classed as flow of control operations.

BCPL's data access revolves around four data areas; global data, static data, local data and immediate data. Immediate data could perhaps be considered as an immutable form of initialised static data. The natural way of modelling the first three data areas is as a base register pointing to the start of a contiguous data area, with an index to reference a particular value. Immediate data is by its very nature encoded within instructions. For local variables the base pointer moves across procedure calls, behaving as a stack. It is interesting to remark that whilst this model is used by most compilers for global and local data, few use it for static data, preferring to compile such code in line, the exceptions being those machines which enforce separate text and data spaces. The scope rules might suggest that grouping all the static variables for a particular routine together is reasonable in order to minimise the offset from a static base register, which would then have to change across procedure call and return. However the scattering of data throughout the code, presumably to be indexed from the program counter seems likely to lead to rather larger offsets.

BCPL's view of data manipulation is very elementary. It uses expressions, which take a number of items from the various data areas and return a result to be used. A simple scratchpad and reverse polish description would at first sight seem to be appropriate as a model. There is certainly a sense in which it is desirable to avoid the use of the BCPL stack for temporary values, since the local stack frame really reflects local variables described by the program. However a little consideration reveals that any scratchpad would have to be arbitrarily large, because of the possibility of recursive functions requiring an arbitrary number of temporary values to be held. Either a compromise, in which temporary variables are saved on the local stack in the event of recursion, or the traditional approach of using the local stack for all temporary results needs to be used. A third approach is a parallel expression stack. However

such a stack is moving away somewhat from the BCPL view of the world, and involves a not insignificant amount of extra work at run time. The whole area of expression handling in BCPL is rather unclear. This is due to BCPL's view of an expression as a black box which takes a number of arguments and returns a result. There is no understanding of the nature of expression evaluation, this is a matter for the machine designer. In any design Knuth's observations on the average size of expressions should be born in mind. Simple expressions must be handled efficiently.

The BCPL view of flow of control would appear to be tree structured. A BCPL program can be viewed as a collection of trees, each tree representing the flow of control in one procedure (this of course does not reflect the scoping rules for variables). Calling a procedure is transfer to the top of another tree, and return the resumption of position in an existing tree. In such an ideal world addresses are virtually redundant, being needed only to describe the transfers between trees. All other control of flow could be done by pushing addresses on to a control stack at the start of constructs, and popping them at the end.

Unfortunately this does not work. BREAK, RESULTIS, LOOP, ENDCASE and GOTO all represent jumps across the tree, rather than walking along it. The first four of these are at least known at compile time, and could in principle be accommodated, however GOTO is evaluated at run time, and its effect on a stack of control addresses in this simple model would be hard to evaluate. Sadly it seems that we have to represent BCPL's tree of control by explicit address markers. In view of the number of flow of control constructs with associated offsets that need to be compiled it is unfortunate that the potential elimination of addresses by use of a control stack cannot be readily achieved. The design of an imperative language in which this could be achieved is a research topic in its own right. Our model of BCPL flow of control will have to use explicit branch addresses. In view of the data on average block sizes given in section 2.2 it seems reasonable that all addressing should be program counter relative.

Having decided on the BCPL view of the world it is now time to choose an instruction set to support it. We start off by introducing the concept of a *canonical* instruction set. This is a minimal instruction set, representing a one to one mapping from each construct in the high level

language to the facilities available in the target low level hardware.

3.2. A Canonical BCPL Instruction Set

If a canonical instruction set is a one to one mapping from the high level instruction set we have at least ensured no additional information is being introduced in bridging the semantic gap. Such an instruction set seems a plausible starting point for a design methodology. The question arises as to what constrains the instructions we chose (after all why not choose the original constructs). At this stage we need to consider the nature of our target architecture (for example CMOS bit-slice, discrete TTL, microprogrammed or interpreted). The instructions we chose must then be implementable "simply" in this target architecture. The definition of "simply" is necessarily vague, and reflects the fact that this stage of the design process is the least regimented. It reflects the idea that instructions should be reasonable within the basic ideas of the architecture. With TTL loading a register is reasonable, evaluating a polynomial is not. Within an interpreter evaluating a polynomial may well be reasonable. There will undoubtably be choices between equally good candidates, and the choice may be influenced ultimately by a background knowledge of what our design criterion is (static size, dynamic size etc.), although at this stage the design criterion is NOT the motivating force. The target architecture in this case was chosen as the High Level Hardware Orion [HLH85]. The Orion is a 32 bit soft microprogrammable mini computer built from standard bit-slice TTL. Support is provided for byte stream instruction sets, with hardware switch on a byte operand provided in the microengine. The example canonical instruction set is therefore a byte stream instruction set. Arguments to opcodes are all 32 bits in length.

Data manipulation is considered as a reverse polish operation using an internal scratch stack. For data access we need a number of operators to push and pop items from this stack, in both word and byte sized chunks, from each of the data areas, and from computed addresses. Table 3.1 summarises the operations provided. There is no distinction of datum size on the internal stack, all entries being word sized. PUSH-type instructions transfer data onto the internal scratch stack, POP-type instructions take data off the internal scratch stack. It is a matter of debate whether strictly there ought to be dyadic PUSH and POP instructions, so that the two

Operation	Data Area				
	global	static	local	immediate	computed
push word	GLOBALPUSH	STATICPUSH	LOCALPUSH	IMMEDIATE	PUSH
push byte	———	———	———	———	PUSHBYTE
pop word	GLOBALPOP	STATICPOP	LOCALPOP	———	POP
pop byte	———	———	———	———	POPBYTE

Table 3.1 Data access instructions

BCPL constructs

!a and a!b

are both supported on a one to one basis. The instructions provided require a!b to be interpreted as !(a+b), introducing an addition operation. With the exception of the instructions which take data from a computed address (PUSH, PUSHBYTE, POP and POPBYTE) all instructions have a single argument. For the IMMEDIATE instruction, this is the datum to load, for the other instructions it is the offset into the relevant data area. The data areas are addressed by three internal registers, G, S and P for global, static and local data areas respectively, and all offsets are in words. PUSH and POP take the word address of their data from the top of the internal scratch stack. PUSHBYTE and POPBYTE take a word address and byte offset from the top of the internal stack. Note that we do not need instructions to push and pop byte sized data from the named data areas, since the only way of referencing byte data is by the computed byte address construct

a % b

Data manipulation operators are straightforward. There is generally one instruction for each BCPL operator, and these are listed in table 3.2. Conditional expressions do not have an operator, but are translated to an equivalent TEST statement to avoid the necessity of evaluating both arms of the conditional. The only problems arise with VALOF blocks and function calls. These can lead to arbitrary growth in the scratch stack by recursion, or undefined

change in its state by jumps out of VALOF blocks. The solution adopted is to cause only the top element of the scratch stack to be valid after a VALOF block or procedure call. This means that the code generator has to create temporary local variables to hold the intermediate values prior to one of these occurring. This is a nuisance, and perhaps the use of an internal scratch stack needs to be reconsidered in the light of the importance of procedure calls. The discussion of CALL, which is really a flow of control instruction will be found below with the description of the other flow of control instructions. With the exception of CALL, none of these instructions have any arguments. Niladic instructions push a value onto the stack (in some sense the instructions to access data are niladic operators), monadic operators replace the top value on the stack and dyadic operators replace the top two values by a single value. The only niladic operator is QUERY, which pushes an undefined value onto the internal stack, corresponding to the ? operator in BCPL. There is one instruction missing, an operator to return the address of an operand corresponding to the address operator @ in BCPL. Once an item is loaded onto the scratch stack it is no longer possible to find its address, and this is thus an operator that works only in combination with instructions that access data. It is in some sense a data access instruction in its own right, and so we need to add to our list of data access instructions the three instructions, GLOBALPUSHLEFT, STATICPUSHLEFT and LOCALPUSHLEFT to push the address of items in these three data areas onto the internal stack.

Flow of control instructions are more complex than the preceding instructions, since it is often necessary to have more than one instruction to map one high level construct, for instance a

Instruction Type	Instructions
niladic	QUERY
monadic	NEG NOT ABS
dyadic	MULT DIV REM PLUS MINUS EQ NE LS GR LE GE LSHIFT RSHIFT LOGAND LOGOR EQV NEQV
function	CALL

Table 3.2 Data manipulation instructions

WHILE needs to be paired with an ENDWHILE to bracket the block on which it operates. A FOR loop needs an instruction to initialise the control variable and an instruction at each end of the loop. Table 3.3 summarises the instructions required for each high level construct. Table 3.4 summarises the meaning of each instruction. In general references to data items are word addresses or offsets, whilst jump addresses and offsets are to byte resolution. A number of points need to be made about the details of the instructions. Results of functions and VALOF blocks are left on the internal stack before executing RESULTIS or RETURN. Various instruction use existing data access and manipulation instructions to initialise values and carry out tests. The start of a WHILE or UNTIL loop is hence not the corresponding WHILE or UNTIL instruction, but the instructions that carry out the loop test immediately beforehand. The FOR instruction has to create an anonymous local variable for its end value, rather than holding it on the scratch stack, lest the value is lost because of a VALOF block or function call. We now have a large number of instructions which map one to one onto our high level language, and all of which are straightforward to implement on an HLH Orion. However a number of these instructions in fact are identical in action, and so the next phase is to eliminate duplicates. For example ENDWHILE and ENDUNTIL are identical, as are IF and WHILE (a WHILE has an ENDWHILE to cause the test to be performed many times). This is only possible for certain flow

Construct	Instructions	Construct	Instructions
procedure call	CALL	TEST conditional	TEST ... ELSE ...
FOR loop	FOR ... ENDFOR	SWITCH conditional	SWITCH ...
REPEATUNTIL loop	... REPEATUNTIL	BREAK statement	BREAK
REPEATWHILE loop	... REPEATWHILE	LOOP statement	LOOP
REPEAT loop	... REPEAT	RESULTIS statement	RESULTIS
WHILE loop	WHILE ... ENDWHILE	ENDCASE statement	ENDCASE
UNTIL loop	UNTIL ... ENDUNTIL	RETURN statement	RETURN
IF conditional	IF ...	GOTO statement	GOTO
UNLESS conditional	UNLESS ...	FINISH statement	FINISH

Table 3.3 Flow of control instructions

Instruction	Meaning
CALL a_1	Call procedure, current stack a_1 words, result left on internal stack
FOR a_1 a_2 a_3	Start of FOR loop, control variable word offset a_1 on local stack, end value offset a_2 on local stack, end of loop at byte offset a_3. Skip to a_3 if loop complete.
ENDFOR a_1 a_2 a_3	End of FOR loop, control variable offset a_1 on local stack, start of loop at offset a_2 backwards, loop increment a_3. Perform increment and loop back.
REPEATUNTIL a_1	End of REPEATUNTIL loop, start at offset a_1. Pop top of internal stack and loop back if value is FALSE.
REPEATWHILE a_1	Ditto, but loop if value is TRUE.
REPEAT a_1	Ditto, but loop back without looking at internal stack.
WHILE a_1	Start of WHILE loop, end at offset a_1. Pop value off internal stack, and jump past end of loop if value is FALSE.
ENDWHILE a_1	End of WHILE loop, which starts at offset a_1 back. Jump back a_1 unconditionally.
UNTIL a_1	As WHILE, but jump if value is TRUE.
ENDUNTIL a_1	End of UNTIL loop, action as ENDWHILE.
IF a_1	Pop value from internal stack; jump forward offset a_1 if value is FALSE.
UNLESS a_1	Ditto, but jump if value is TRUE.
TEST a_1	Identical to IF, but forward jump is to point immediately after ELSE.
ELSE a_1	Unconditional jump forward a_1.
SWITCH $a_1, \ldots, a_n$	Perform SWITCH. a_1 is offset for DEFAULT, a_2 is number of cases. Other arguments are pairs of value and offset for each CASE.
BREAK a_1	Jump out of loop. Unconditional branch forward a_1.
LOOP a_1	Jump to end of loop. Unconditional branch forward a_1. Really needs LOOPBACK as well for efficiency.
RESULTIS a_1	Unconditional jump forward a_1 out of VALOF block.
ENDCASE a_1	Unconditional jump forward a_1 out of SWITCH block.
RETURN	Return from procedure. Absolute branch address given in stack frame.
GOTO	Jump to absolute address on top of internal stack.
FINISH	Terminate program.

Table 3.4 Meaning of Instructions

of control instructions in this instruction set, but there is no reason why such an effect would not be seen with other classes of instruction under different circumstances. It does not eliminate the one to one mapping, since there is still no introduction of redundant information in the translation. We end up with the flow of control instructions CALL, FOR, ENDFOR, REPEATUNTIL, REPEATWHILE, JUMP, REPEAT, RETURN, GOTO, FINISH, IFWHILE, UNLESSUNTIL, SWITCH. Table 3.5 is a revised version of table 3.3 showing how these unique flow of control instructions are used.

We now have a set of 48 canonical instructions and can write a compiler. The standard way of building a new compiler for BCPL is to write a code generator for the intermediate code produced by the front end, OCODE. However OCODE, although a reasonable approximation to the BCPL world has already imposed some of its own ideas, and is not therefore the best way of generating a new canonical instruction set. A far better source for the code generator is the AE-tree produced by the syntax analyser, which is a close representation of the high level structure of the original BCPL program. Such a compiler for the canonical instruction set was written for this project. The instructions of the canonical instruction set correspond very closely to the node types on the AE-tree, and code generation is relatively easy. Constant folding and

Construct	Instructions	Construct	Instructions
procedure call	CALL	TEST conditional	IFWHILE ... JUMP ...
FOR loop	FOR ... ENDFOR	SWITCH conditional	SWITCH ...
REPEATUNTIL loop	... REPEATUNTIL	BREAK statement	JUMP
REPEATWHILE loop	... REPEATWHILE	LOOP statement	JUMP or REPEAT
REPEAT loop	... REPEAT	RESULTIS statement	JUMP
WHILE loop	IFWHILE ... REPEAT	ENDCASE statement	JUMP
UNTIL loop	UNLESSUNTIL ... REPEAT	RETURN statement	RETURN
IF conditional	IFWHILE ...	GOTO statement	GOTO
UNLESS conditional	UNLESSUNTIL ...	FINISH statement	FINISH

Table 3.5 Flow of control instructions (revised)

common sub-expression elimination can be carried out on the AE-tree, and amount effectively to source code optimisations. We are thus working with optimised source code as recommended by Flynn [Flynn84]. The ease with which instructions match up to the AE-tree is an indication that our canonical instruction set is a valid one to one mapping.

To test the compiler, and if dynamic statistics are wanted, an interpreter needs to be written. This is a far from trivial task, since although interpreting the instructions is easy, we also need to interpret Tripos system calls if the sample programs are to be properly evaluated. This is not helped by 3000 lines of the Tripos kernel being written in assembler. A simple interpreter was built, and sufficient interface provided to show that simple Tripos commands would compile correctly, an indication that the compiler is unlikely to be too wildly inaccurate. However the implementation of sufficient operating system to enable all 102 test programs to run and produce useful data was deemed to be unnecessary. The prime interest is in static code size, and dynamic statistics are not of immediate importance.

This experimental compiler was used to obtain static code statistics on the 102 BCPL programs from the TRIPOS command environment. These generated 520 053 bytes of code. That this is substantially larger than the figures given for other machines above is not worrying, since at this stage all instructions have explicit arguments which are 32 bits long. If anything it is surprising that it is only 21% larger than equivalent code for the IBM S/370. This is the equivalent of Sweet and Sandman's normalised code [Sweet82]. We must now look at how to refine this instruction set to obtain a version that meets our criterion of static compactness.

3.3. Design Rules for an Instruction Set

New instructions may be derived from those provided in the canonical instruction set in a number of ways. Such methods of deriving new instructions from those existing already are called "design rules" Three methods can be found explicitly in the literature and are used by Sweet and Sandman for MESA, and Tanenbaum for EM-1. These design rules are

(1) Create a new instruction with a smaller argument. This can then be used for all the cases of the old instruction where the argument would fit in the new argument size. For example (GLOBALPUSHBYTEARG1) could be derived from GLOBALPUSH to push global variables 0-255 onto the scratch stack.

(2) Create a new instruction with a single value of one argument implied. Thus (IMMEDIATEARG1=0) could be derived from IMMEDIATE, to push constant zero onto the scratch stack.

(3) Create a new instruction by concatenating two existing instructions. For example GLOBALPUSH and CALL could be combined to give (GLOBALPUSHCALL), to call a global routine. The arguments to each of the original opcodes become arguments to the new opcode in the order of their corresponding opcode.

Derived instructions are added to the pool of instructions, and the rules may then be applied to these new instructions. For example we may get ((GLOBALPUSHARG1=73)(CALLBYTEARG1)) to call global routine 73 (which under Tripos is the routine WRITES, which writes out a string) with a stack frame offset in the range 0-255.

These are far from being the only rules that could be used. Rule 2 could be generalised so that new instructions with arguments of a limited range, not necessarily starting from zero could be created. We could then derive an instruction to push globals in the range 256-511 for example. Such a design rule is effectively used in the derivation of CINTCODE described earlier in section 1.5. Wade and Stigall's work with the entropy of instruction sets [Wade75] justifies the splitting up of instructions into streams of simpler instructions. For example FOR and ENDFOR could be replaced by IFWHILE and REPEAT with some initialisation instructions. UNLESSUNTIL could be replaced by NOT and IFWHILE. Such a rule would require some initial information on rewriting instructions in the canonical instruction set. A simpler version of this rule would be to remove any instruction that becomes completely unused. The number of rules that may be derived is limited by the ingenuity of the designer, although the potential gain from the more obscure rules is small. The only constraint is the same as that placed on the original choice of instructions. The rules must lead to instructions that can be implemented "simply" using the target architecture.

The names of the generated instructions are designed to give a guide as to the formation of the instruction. They are constructed as follows:

(1) When an argument is shrunk, the suffix BYTEARGx, HALFWORDARGx or THREEBYTEARGx is added, where x is the number of the argument involved.

(2) When an argument is implied within an instruction, the suffix ARGx=y is added, where x is the argument involved, and y the value being combined.

(3) When two instructions are combined, the new name is the concatenation in order of their two names.

Each new instruction name is then surrounded by brackets to avoid ambiguity. The names are not perfect. In particular the use of rule three leads to wrong numbering of arguments; the numbering refers to the original position of arguments within instructions, and the position in new instructions must be resolved by observation. Such names are inherently verbose but it is anticipated that once an instruction set has been finalised more concise mnemonics for the instructions would be provided.

To decide which rule to apply statistical data on the compiled code will be required. The first analysis to be carried out is to look at instruction distribution. Table 3.6 shows the static frequency[7] distribution of instructions in the compiled code to the nearest 0.01%.

One simplification has been made to the statistics as used. SWITCH is the only instruction to have a variable number of arguments. To simplify the analysis at this stage, the value-offset pairs that comprise the bulk of a SWITCH instruction have been removed, some 20768 bytes of data, which would have increased the frequency of the SWITCH instruction to 4.4%, and decreased other frequencies by a factor of about 0.96. The data as presented represent some

[7] *Frequency* is a rather ambiguous word in this context. It does not mean the number of times a particular instruction occurred as a percentage of all instruction occurrences, but is rather the amount of space occupied by a particular instruction as a percentage of the space occupied by all the compiled instructions. This means that instructions with several arguments (e.g. FOR, SWITCH) will be more prominent than those with no arguments (for example the operators). Since we are interested in the minimisation of code size this is the definition of frequency that is most meaningful, and unless otherwise indicated such weighted frequencies are used throughout the rest of this chapter.

Instruction	Frequency	Instruction	Frequency	Instruction	Frequency
IMMEDIATE	20.23%	PUSH	0.66%	POPBYTE	0.06%
LOCALPUSH	17.75%	LOCALPUSHLEFT	0.42%	LS	0.05%
GLOBALPUSH	15.17%	SWITCH	0.41%	REPEATWHILE	0.04%
CALL	9.92%	RETURN	0.34%	MULT	0.04%
LOCALPOP	9.79%	POP	0.27%	NOT	0.04%
IFWHILE	4.42%	LOGAND	0.23%	GE	0.03%
JUMP	3.76%	MINUS	0.18%	DIV	0.03%
GLOBALPOP	3.21%	NE	0.16%	RSHIFT	0.03%
STATICPUSH	2.48%	LOGOR	0.15%	LSHIFT	0.02%
STATICPUSHLEFT	2.38%	PUSHBYTE	0.13%	QUERY	0.02%
FOR	1.72%	STATICPOP	0.11%	REM	0.01%
ENDFOR	1.72%	LE	0.10%	NEG	0.01%
PLUS	1.21%	GLOBALPUSHLEFT	0.08%	ABS	0.01%
UNLESSUNTIL	1.03%	GR	0.07%	FINISH	0.00%
EQ	0.74%	GOTO	0.07%	NEQV	0.00%
REPEAT	0.67%	REPEATUNTIL	0.06%	EQV	0.00%

Table 3.5 Canonical Instruction Set Frequency Distribution

499285 bytes of compiled code, with the SWITCH instruction transformed into a two argument instruction. This is a simplification that will be considered in the final section of this chapter when the analysis is extended.

It is striking the frequency distribution is dominated by just a small number of instructions. All instructions were used, but some extremely rarely. EQV occurred twice - 0.0004% of the compiled code. The first 10 instructions take up almost 90% of the compiled code space. However this should not be a cause for surprise, since these are all instructions that are likely to be used in the compilation of assignment, procedure call and conditional statements, which were shown in section 2.2 to comprise nearly 83% of all statements.

If we follow Sweet and Sandman's approach we should also obtain histograms of argument distributions, and of pair frequencies. Such statistics were obtained, but are verbose in the extreme. They may be obtained in machine readable form from the author on request. It should be noted when considering argument distributions that idealised stack offsets as described in the section 2.2 have not been implemented. Given the nature of the design rules, which favour instructions in which particular argument values are very frequent this is an improvement which should be considered for future refinement of the system. Given the volume of data it is difficult to observe particular items of interest in the pair frequency information. However it is notable that the second commonest pair is PLUS followed by PUSH, suggesting that the failure to include dyadic PUSH and POP instructions was a mistake.

Using these rules we may generate an arbitrary number of instructions. The number is limited initially in this example to 256 instructions, although with escape opcodes there is no reason why more instructions may not be used, as in EM-1. The major advance in this new design methodology over previous work is in the way in which new instructions are chosen. We consider which rule, applied to which opcode(s) and argument(s) will lead to the greatest improvement in the design criterion, in this case static code size. To do this a program called ISGEN, an instruction set generator has been developed. The next section describes this system and considers whether the statistics on compiled code derived above are adequate to select the correct instruction. In section 5.2 below the theoretical justification of this methodology is considered.

3.4. ISGEN

ISGEN works by taking a set of statistics on a given instruction set, and considering which rule would lead to the greatest saving in size. The design rules are implemented as pairs of C procedures, one to determine the best saving in static space that can be obtained with the rule, the second to create an information node for a new instruction created by that rule and then deduce statistical information for it. This process is repeated until the required number of instructions have been created, in this case 256.

Statistical information for each instruction is held in a linked structure as shown in figure 3.1. By chosing a fairly general linked structure of this form it is possible to extend the sort of information held in future versions of ISGEN with a minimum of disruption. The name of the instruction is held as a tree structured node, to facilitate the construction of new instruction names in the manner described in the previous section. The size of an instruction is the size of its opcode and the sum of the sizes of its arguments. For convenience the total size of the instruction as well as the size of the opcode itself is held in the opcode part of the data structure. The frequency held in the opcode node is a count of the occurrence of the instruction, i.e. not weighted by instruction size. In general frequency information held in the data structure is *not* weighted by instruction (or opcode, or argument) size, it being more convenient to carry out weighting explicitly when necessary. The information nodes for each argument identify which

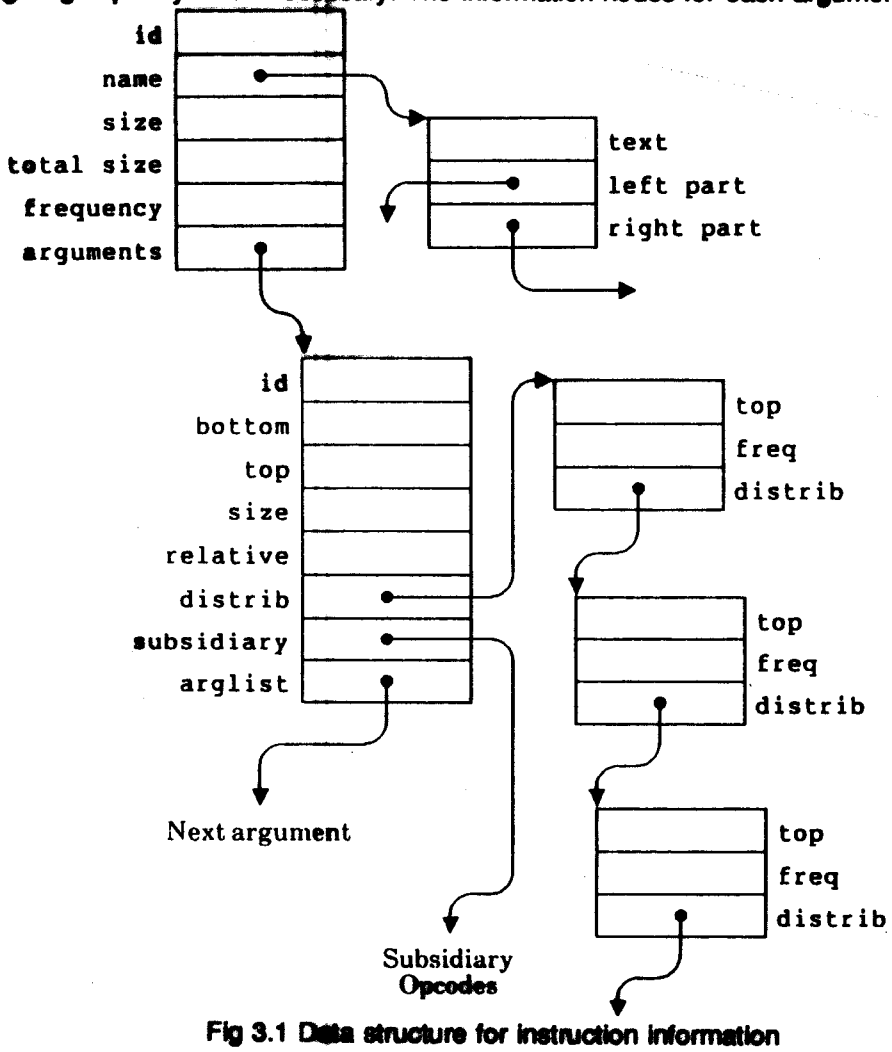


Fig 3.1 Data structure for instruction information

argument it is, and give a range of data it may hold, as well as the size of the argument (really only two of these three items are needed, but is convenient to have all three to hand). The *relative* field marks whether this argument is a program counter relative offset. If it is such an argument then its value is likely to change as the size of compiled code as a whole shrinks. The distribution of argument values is represented as a linked list, representing a histogram of argument values. The subsidiary field points to a related instruction with narrower specification. This is to cope with the case where shrinking code means more of a relative argument can fit in a smaller size. For example many of the argument values used with a JUMP instruction will be too large to fit in the argument to a JUMPBYTE opcode. However by the end of instruction generation, when code size, and hence offsets are substantially smaller, far more will fit. All frequencies are done as unweighted integer counts of instructions, rather than as fractions of the total sample code size. This makes for faster arithmetic, and ensures rounding errors are noticed. A preliminary version of the program used floating point, expressing instruction frequencies as a fraction of the total code size. It was very difficult to distinguish between systematic errors due to flaws in the algorithm (such as fence-post problems with the last argument value in a new instruction) and general rounding errors due to the large amount of floating point arithmetic being used. With integer arithmetic, and frequencies held as number of bytes of a particular instruction there are no rounding errors. Any systematic errors show up as the loss of a complete instruction from the sample data and are readily monitored. Given the large volume of data, and the non-trivial nature of some of the calculations this is a useful guard against mistakes.

For each rule all possible opcode/argument combinations are tried, and the statistics used to estimate the saving that would accrue by applying the rule to this combination. The combinations that give the best result from each rule are compared, and the best of those selected. This is the opcode that will next be created. An information node is created for the new opcode, and estimates are made of the statistical information about it. The statistical information about existing instructions is adjusted correspondingly.

We need to consider how the statistical information is used for each of the rules I have described, and how statistical information about the use of new instructions proposed can be

generated.

In the case of the rule to shrink an argument the amount of space saved in bytes is given by

$$n \times (os - ns)$$

where n is the number of existing examples of this instruction for which the argument concerned has a value that could be handled by the new instruction. This is found by counting down the distribution histogram finding all the argument values that are small enough to fit in the new instruction. *os* is the size of the existing argument under consideration, *ns* is the size of the corresponding argument in the new instruction.

It is easy to calculate the statistics for the new instruction if the existing instruction has only one argument, since in this case that part of the distribution list for the argument that contains values small enough to fit in the new instruction argument goes to the new instruction, and the remainder stays with the existing instruction. However if there are other arguments we need to consider how their distributions are affected. ISGEN assumes that the distribution of other arguments is independent of the argument under consideration. Some of the experimental results given in the next section underline the fact that this is not always a valid assumption. There is precedence for the assumption; it was used by Wade and Stigall in calculating opcode entropies [Wade75]. Consideration of the theoretical significance of this assumption is given in section 5.1 below. As in the single argument case such additional arguments may be partitioned between the new and existing arguments. Other arguments have their frequency distributions partitioned element for element in the same ratio. Care is taken to ensure that rounding errors are propagated, so that the total occurrence of each argument for a given opcode is the same.[8] Handling the relationship between instruction pairs is non-trivial. Again the assumption is made that the probability of any instruction occurring is independent of its neighbouring instructions. Using this the relationship between the new instruction and any other instruction and the existing instruction and any other instruction can be calculated as fractions of the relationship to the existing instruction.

---

[8] If an instruction occurs 5 times, clearly each of its arguments must also occur 5 times. The only exception is if the instruction has a variable number of arguments, e.g. the SWITCH instruction.

$$P(n, x) = P(o, x) \times \frac{nf}{of + nf}$$

$$P(x, n) = P(x, o) \times \frac{nf}{of + nf}$$

$$P(o, x) = P(o, x) \times \frac{of}{of + nf}$$

$$P(x, o) = P(x, o) \times \frac{of}{of + nf}$$

where

o is the existing instruction

n is the new instruction

nf is the frequency of the new instruction deduced from argument splitting

of is the revised frequency of the existing instruction

$P(x, y)$ is the frequency of the x,y instruction pair

All frequencies are unweighted.

These equations are fine, so long as the frequencies involving the new instruction are evaluated first. All values involving the new instruction are derived in terms of frequencies involving the old instruction, before such frequencies are themselves adjusted. In particular $P(n, n)$ is derived as the original $P(o, o)$ twice multiplied by the appropriate factor. This is a prime case where checking that the frequencies total the same before and after will avoid errors in calculation.

An almost identical process is involved in creating an instruction with a specific value of argument implied, indeed it is akin to creating an opcode with a zero sized argument. There is a problem with relative address arguments, which will clearly change in value as generation progresses, and instructions become smaller and move together. However at present ISGEN ignores the problem, by not applying this rule to relative arguments. This is perhaps not the correct approach, since as code shrinks we would expect arguments of the same value to be affected equally implying that the instruction would quite likely still be justified, but with a different argument value. Given that the savings due to this rule are not as great as for the shrinking rule, and the rather wide spread of relative arguments anyway the error in making this assumption would be quite small. In the next section an estimate of the size of this error is

given.

The rule that combines instructions creates some different problems. To estimate the saving from the statistics is easy and accurate, since the space required by the new instruction will be the same space for all the arguments, and space for one opcode rather than two. Assuming all opcodes are one byte the saving is the unweighted frequency of the given instruction pair as a number of bytes.

Generating argument distributions for the new instruction makes the same assumption of argument independence as before. Argument lists are reduced in frequency according to the unweighted frequency of the new instruction, and the remainder given to the new instruction. Calculation of the new instruction pair frequencies is similar to the earlier examples and is given by the equations:

$$P(n, x) = fn2 \times P(o2, x)$$

$$P(x, n) = fn1 \times P(x, o1)$$

$$P(o2, x) = fo1 \times P(o2, x)$$

$$P(x, o1) = fo2 \times P(x, o1)$$

Where:

$$fo1 = \frac{P(o1) - P(o2)}{P(o1)}$$

$$fo2 = \frac{P(o2) - P(n)}{P(o2)}$$

$$fn1 = \frac{P(n)}{P(o1)}$$

$$fn2 = \frac{P(n)}{P(o2)}$$

n	is the new instruction
$o1$	is the first original instruction
$o2$	is the second original instruction

x is an arbitrary instruction

$P(x)$ is the frequency of instruction x (before n was created)

$P(x1, x2)$ is the frequency of instruction pair $x1$, $x2$

Once again frequencies involving the new instruction are calculated first. Again cross checking that the total of all the frequencies is unchanged is important.

3.5. Results

The evaluation of the results from ISGEN was carried out with the aid of a simple peephole optimiser, which added the new instructions to the existing code. The peephole optimiser takes data from ISGEN giving details of the instructions that were generated, and substitutes each instruction in turn, thus copying the behaviour of the generator. This is not a complete peephole optimiser, in that it does not consider adding all the instructions at once, and will not always use

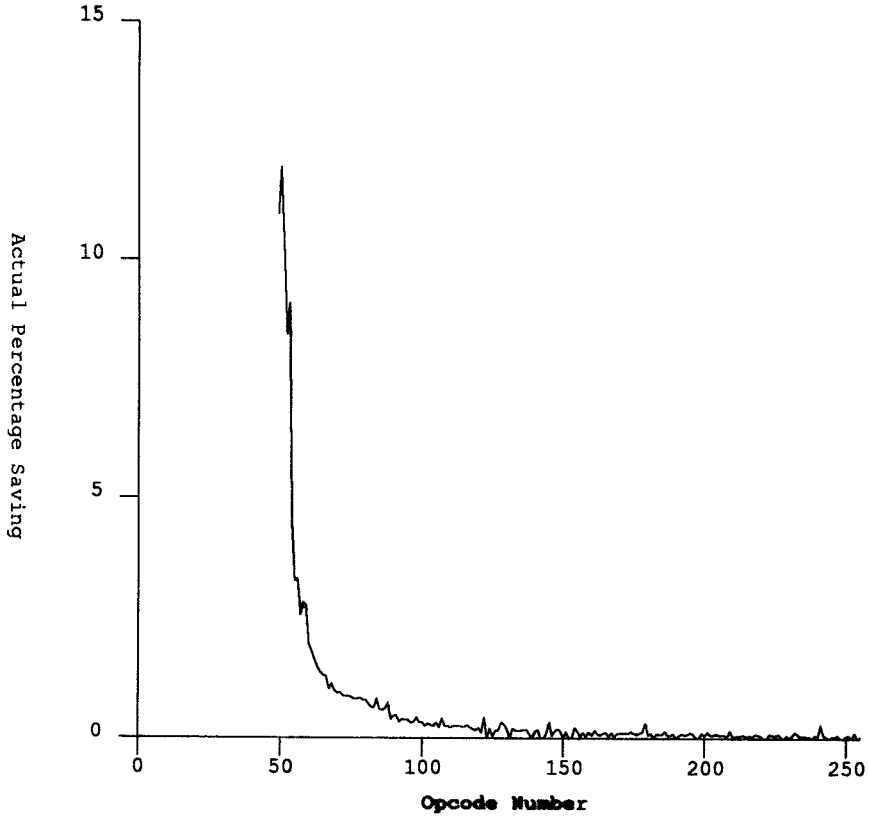
New Instruction			
#	Name	Individual	Cumulative
49	(IMMEDIATEBYTEARG1)	89.04%	89.04%
50	(LOCALPUSHBYTEARG1)	88.04%	78.39%
51	(GLOBALPUSHBYTEARG1)	89.69%	70.31%
52	(CALLBYTEARG1)	91.54%	64.36%
53	(LOCALPOPBYTEARG1)	90.88%	58.49%
54	(IFWHILEBYTEARG1)	95.61%	55.92%
55	(GLOBALPOPBYTEARG1)	96.74%	54.10%
56	(JUMPBYTEARG1)	96.70%	52.31%
57	((GLOBALPUSHBYTEARG1)(CALLBYTEARG1))	97.27%	50.88%
58	((IMMEDIATEBYTEARG1)ARG1=0)	97.19%	49.46%

Table 3.6 The first few ISGEN results

the best instructions for substitution. However it is helpful in that it allows us to look at how individual predictions have fared. A typical listing from ISGEN is shown in Appendix A, and the first few results are given in table 3.6. The results are given as percentage reduction in total code size, both individually and cumulatively. These results are perhaps seen more clearly graphically. Graph 3.1. is a graph of predicted saving (note this is 100% - the figure for reduction above) against instruction number. Instruction numbers are assigned in sequence, so this gives the order in which the instructions were found. The final predicted reduction in size was to 28.16% of the original size. The jagged nature of graph 3.1 might seem a contradiction, since for example it seems that instruction 50 contributes a greater saving than instruction 49 (11.96% for instruction 50 against 10.96% for instruction 49), and hence ought to have been generated first. However this is because these results are given as percentage savings over the size of the code after the previous reduction, and thus the absolute saving in code size is in fact less (10.65% of original size for instruction 50 as opposed to 10.96% for instruction 49).

Looking at the actual instructions generated we see they fall very much in the pattern expected. The first two and the fifth are associated with assignment to local variables, the third and fourth with procedure calling, and the sixth instruction with conditional branching. The lack of orthogonality is also striking, in contrast to many modern processor designs. Only those instructions actually needed are generated.

The peephole optimiser was applied to try out the suggested new instructions as given by ISGEN. Table 3.7 shows the savings obtained with the peephole optimiser for the first few instructions compared with the predicted values obtained using ISGEN. From this table it would seem that predictions are remarkably accurate, and overall this would seem to be true, with a predicted code size of 28.16%, and an actual achieved code size of 28.53% compared to the original code size. However there is a need to look closely at the results of optimisation. There are a number of individual instructions where the predicted and peephole savings differ significantly. Admittedly most of these examples occur late on in the generation and their effect in absolute terms is not large, however it is important to examine the causes behind such erroneous predictions, lest they reveal a flaw in the system as a whole. Of particular interest are the fourteen instructions in table 3.8 which although predicted to make savings, in fact are



Graph 3.1 Code Size Saving v Instruction Number

New Instruction			
#	Name	Peephole	Predicted
49	(IMMEDIATEBYTEARG1)	10.96%	10.96%
50	(LOCALPUSHBYTEARG1)	11.96%	11.96%
51	(GLOBALPUSHBYTEARG1)	10.31%	10.31%
52	(CALLBYTEARG1)	8.46%	8.46%
53	(LOCALPOPBYTEARG1)	9.12%	9.12%
54	(IFWHILEBYTEARG1)	4.39%	4.39%
55	(GLOBALPOPBYTEARG1)	3.26%	3.26%
56	(JUMPBYTEARG1)	3.30%	3.30%
57	((GLOBALPUSHBYTEARG1)(CALLBYTEARG1))	2.56%	2.73%
58	((IMMEDIATEBYTEARG1)ARG1=0)	2.80%	2.81%

Table 3.7 Peephole and Predicted Savings from ISGEN

never used. Most interesting is the case of (((IMMEDIATEBYTEARG1)ARG1=0)MINUS). In this case the first generations to (IMMEDIATEBYTEARG1) and then to ((IMMEDIATEBYTEARG1)ARG1=0) are reasonable enough. However our initial data on instruction pairs had IMMEDIATE followed by MINUS a very common pair. Reasonable enough, since subtracting a constant is a common operation. Our data on IMMEDIATE showed that zero was its commonest argument. Our assumption of independence of instructions meant that in deriving information about IMMEDIATE followed by MINUS we assume that the most common IMMEDIATE value to be subtracted is zero. Only common sense tells us otherwise.

((GLOBALPUSHBYTEARG1)ARG1=74) illustrates another problem. Global value 74 is under TRIPOS the very common routine WRITEF, and it is reasonable that we should have an instruction to load this value. However it is always followed by a CALL instruction, and by the time we come to generate ((GLOBALPUSHBYTEARG1)ARG1=74) all occurrences of GLOBALPUSHBYTE with argument value 74 have already been subsumed into ((GLOBALPUSHBYTEARG1)CALLBYTEARG1) and ((GLOBALPUSHBYTEARG1)CALL).

New Instruction		
#	Name	
123	((JUMPBYTEARG1)RETURN)	0.22%
131	(((LOCALPOPBYTEARG1)(LOCALPUSHBYTEARG1))ARG1=0)	0.19%
143	((JUMPBYTEARG1)((IMMEDIATEBYTEARG1)ARG1=0))	0.15%
153	((LOCALPOPBYTEARG1)(((FORBYTEARG1)BYTEARG2)BYTEARG3))	0.13%
190	(((IMMEDIATEBYTEARG1)ARG1=0)MINUS)	0.09%
221	((JUMPBYTEARG1)((STATICPUSHLEFTHALFWORDARG1)BYTEARG1))	0.07%
222	(((LOCALPOPBYTEARG1)ARG1=0)(((FORBYTEARG1)BYTEARG2)BYTEARG3))	0.07%
228	((GLOBALPUSHBYTEARG1)ARG1=74)	0.07%
230	((JUMPHALFWORDARG1)RETURN)	0.07%
238	((JUMPBYTEARG1)((LOCALPUSHBYTEARG1)ARG1=0))	0.06%
244	((GLOBALPUSHBYTEARG1)ARG1=73)	0.06%
249	(((LOCALPOPBYTEARG1)ARG1=1)(((FORBYTEARG1)BYTEARG2)BYTEARG3))	0.06%
252	((JUMPBYTEARG1)(LOCALPUSHBYTEARG1))	0.06%
254	((((LOCALPOPBYTEARG1)(LOCALPUSHBYTEARG1))ARG2=1)ARG1=0)	0.06%

Table 3.8 Predicted Savings with no Peephole Saving

It is clear that our statistical data is not complete enough to cope with these interdependencies. This is an example of the standard result from Operations Research that local optimisation does not necessarily lead to global optimisation.

The system as described uses three rules. Adding more rules would appear to lead to diminishing returns. The results of using one two and three rules are summarised in Table 3.9. The rule to reduce the size of an argument is clearly the most beneficial. However this must to a certain extent be a reflection of the use of 32 bit arguments throughout the canonical instruction set. More important is the non-cumulative nature of the results. This indicates that rules interact to some extent.

Rules	Predicted Saving
Shrink argument (S.A.)	44.19%
Combine argument and opcode (C.A.)	51.19%
Combine instruction pairs (C.I.)	87.22%
S.A. and C.A.	31.74%
S.A. and C.I.	32.56%
C.A. and C.I.	53.31%
S.A., C.A. and C.I.	28.16%

Table 3.9 The effect of using various rules

The original discussion of the nature of BCPL came to the conclusion that instructions fell into three basic groups; data access, data manipulation and flow of control. The canonical instruction set contains respectively 14, 21 and 13 instructions in each group. In compiled code these groups occur with (unweighted) frequency 73%, 3% and 24% respectively. The question arises of what proportion of the final instruction set should support each group. If we are to take the same view as Richards did with CINTCODE [Richards84], then we should propose that our final instruction set reflect the distribution of instructions within compiled code. In other words the final instruction set should be 73% data access instructions, 3% data manipulation and 24% flow of control. Table 3.10 shows the observed results in using the ISGEN instruction set (the fractional instruction counts reflect instructions combined from two groups). Leaving aside the fact that with 21 canonical data manipulation instructions we could not hope to have less than 8% of such instructions in the final instruction set, these do not bear out Richards' hypothesis. In section 5.3 below we consider whether this result is to be expected when generating an optimal instruction set, and the validity of Richards' hypothesis.

In the light of all these results we need to consider the degree of success in selecting instruction sets achieved by ISGEN. We have seen that the statistics used along the lines suggested by Sweet and Sandman are not totally adequate, in that they cannot express global dependencies. However as a first approximation they work reasonably well, the worst error observed being a

Group	Instruction Set		Compiled Code	
	#	%	#	%
Data Access	150.970	59%	363068	73%
Data Manipulation	35.814	14%	15609	3%
Flow of Control	69.756	27%	120608	24%
Total	256.000	100%	499285	100%

Table 3.10 Distribution of instruction types

predicted 0.22% reduction that did not materialise, at a time when the code had already been reduced to 32.25% of its original size, an overall error of 0.08% of the original size. Such errors are due to the use of deduced "statistics." Much better would be to perform peephole substitution as each instruction is created and then regenerate the statistics from the substituted code. This would lead to an increase in running time, which at present is about 10 minutes on an unloaded HLH Orion, to about an hour, roughly the time the current peephole optimiser takes.

The use of a greedy algorithm, taking the best instruction each time has not been justified, and this is considered in section 5.2 below. However the evidence of interacting rules presented above might indicate that better algorithms could be used. Some form of lookahead, to estimate at least local interdependencies might lead to improvement in the quality of instructions generated, although the computing power required could be increased dramatically by such algorithms. The current algorithm is simplistic in a number of respects, notably in its handling of relative arguments. The exclusion of such arguments from the combine argument rule was mentioned in the previous section. To release this restriction is not difficult, but as suspected it makes little difference, with a predicted reduction in code size of 28.08% as opposed to the 28.16% with the restriction imposed. More significant is the way shrinking of code is not compensated for when evaluating the savings due to instructions with reduced argument sizes, as in the case of (JUMPBYTEARG1) for example. With code size reduction by a factor of nearly

four, offsets of up to around 1000 when using the canonical instruction set would fit in a one byte argument at the end of instruction generation. Fortunately some approximate investigation suggests that this is not as much of a problem as might be supposed. The biggest difference is with(JUMPBYTEARG1). Without allowing for shrinkage this is responsible for a saving of 3.30%. Allowing for shrinkage this saving would go up to around 3.83%, at a stage when the code size has already been shrunk by nearly 50%. In total by allowing for shrinkage the predicted code compaction is estimated to go up to 27.69%, from 28.16% without allowing for shrinking relative arguments. It would certainly seem this is not an error worth serious worry.

Another point worth noting that can be seen from the full listing in appendix A is that 256 instructions are not really needed. Over 90% of the saving in space is achieved by the first 41 new instructions created, a total of 89 instructions. The last 43 instructions contribute less than 1% to the saving. This places something of a question mark over those byte stream instruction sets that make use of an escape opcode. A further aspect of this particular point is to note the need for a rule to eliminate redundant instructions. Some of the canonical instructions are totally subsumed into new instructions. Even some of the new instructions generated early on are taken up completely by later instructions. Whilst there might be a requirement to keep some redundant instructions for system completeness, there is clearly a need to remove some. This reduces further the number of instructions required for a successful compact instruction set.

Finally we need to look at the degree of success achieved by ISGEN. Table 3.11 is a revised version of table 2.9 and shows the size of compiled code for the 102 sample programs for various target instruction sets, including the canonical instruction set and the new instruction set proposed by ISGEN. The results have been adjusted to allow for static string data and the SWITCH jump tables that were removed before analysis. Once again results are relative to OCODE. Considerable success has been achieved. The compiled code, even with 12.5% static string data is around half the size of code for popular existing instruction sets, including a modern "high level" instruction set, the MC 68000. Even more pleasing, compiled code for the new instruction set is 14% smaller than compiled CINTCODE, itself designed as a compact instruction set. Clearly mechanised design can improve over human experience.

Instruction Set	Code Size (bytes)	Size Relative to OCODE
OCODE	302273	100%
New Instruction Set	207510	67%
CINTCODE	240000	79%
MC 68000	385936	128%
IBM S/370	465000	154%
Canonical Instruction Set	564349	187%

Table 3.11 Compiled code size for BCPL using different instruction sets

This example has shown how a complex program, representing many man-months work can automatically refine an instruction set for BCPL. To be useful such a program must be more general, capable of dealing with other languages, meeting other design criteria, and dealing with more complex systems. In the final section of this chapter we look at ISGEN applied to a different language system, and consider how well it can handle different design criteria, or more complex systems.

3.6. Extensions to ISGEN

ISGEN as it stands is already general enough to deal with other languages implemented via byte stream instruction sets. It was used to refine an instruction set for the polymorphic programming language, POLY [Matthews85]. This uses a 16 bit byte stream instruction set as an intermediate code output by the compiler front end. Matthews wished to refine this to reduce the space occupied by this intermediate code. It was hoped that the resultant instruction set would also be suitable for microcoding as a machine to run POLY directly. In this example ISGEN was used to refine the existing instruction set, along the lines used by Sweet and Sandman with MESA. The existing intermediate code, with 24 instructions was taken as the canonical instruction set. The same three design rules as applied with the experimental BCPL instruction set were used. Sample statistics were provided from 214074 bytes of compiled code.

Appendix A includes the output from the run of ISGEN on this data. A reduction in code size to 38.46% of the original size was predicted. That this reduction in size is not as great as that found with the BCPL canonical instruction set is not surprising. It is an effect of working with a 16 bit instruction set, so that there are not so many large savings due to common instructions with large arguments being almost totally replaced by instructions with small arguments. This is reflected in the fact that to achieve 90% of the total saving required the generation of 90 new instructions (a total of 114 instructions), compared with only 41 new instructions for BCPL.

Matthews used this proposed instruction set in a rather different way to the instruction set proposed for BCPL. He accepted only the first 97 proposed instructions, responsible for around 88% of the improvement and incorporated them into his compiler, rather than using peephole substitution. This permitted him to take advantage of global interaction between instructions, which was seen to be a disadvantage with straightforward peephole optimisation. He then used his new compiler to generate further statistics. The new compiler produced 82560 bytes of compiled code, a reduction to 38.57% of the original code size. This suggests that there is a not insignificant gain to be achieved from global interaction of new instructions. These new statistics were then fed back into ISGEN, which proposed a further refinement of the instruction set to achieve a predicted reduction in size to 74.87%. These revisions have yet to be used, but even allowing for no gain due to global interaction an overall reduction to 29% of the size of the original code seems plausible. This is rather more impressive than the reduction to 28% achieved with BCPL in that it was achieved not over an artificially verbose 32 bit canonical instruction set, but over an existing 16 bit instruction set.

To develop instruction sets to support an individual language is not always good enough. For example the HLH Orion on which much of this work has been carried out is regularly used for running LISP, C, BCPL and FORTRAN. One approach already mentioned is to use machines with a number of alternative instruction sets, swapping in the required instruction set as necessary. Indeed the HLH Orion is one of the few machines that permit this mode of operation. However such instruction set swapping does represent one more run time overhead, which it would be helpful to avoid. I suggest that such an approach is heavy handed. Both the examples presented obtained most of their gains with less than half the available 256 instructions. A single

byte stream instruction set could incorporate both with almost all the gain found with the individual instruction sets. Such an idea is not new. The IBM S/360 contained the instruction set from earlier commercial machines, to support COBOL and from earlier scientific machines, to support FORTRAN. It is likely that many instruction sets could be incorporated into a single byte stream instruction set because of overlap of instructions. For a single language system we design a canonical instruction set for one language, and merge instructions (such as IF and WHILE with BCPL) which are semantically identical. For a multiple language system we design a canonical instruction set for each language, and then merge instructions across instruction sets that are semantically identical. In a multi-language system there is likely to be considerable common ground, not only in simple arithmetic operations, but also with higher level instructions. It is highly likely that in sharing a system languages will have to share a stack structure to some extent and obey linking conventions. Having designed a common canonical instruction set compilers can be written for each language and used to obtain a combined statistical sample on programs to be run in the target environment. ISGEN will then yield an optimal common instruction set. There are greater constraints with this type of design. It would not be particularly amenable to design of one instruction set from scratch (as was done for BCPL) and refinement of another existing instruction set (as was done for POLY), because of the lack of similarity between the canonical instruction sets.

ISGEN as developed optimises instruction sets with respect to static size of compiled code. However it is clear that any criterion of optimisation can be used subject to two conditions:

(1) The criterion must be quantifiable, in order that we can distinguish which rule applied to which opcodes and arguments will lead to the greatest saving with regard to the criterion.

(2) It must be possible to obtain statistics for derived instructions from the available data. For static code size this is straightforward. For dynamic code size this might mean running a simulator on the new code, or performing fairly complex flow analysis.

The design criteria need not be so prosaic as these. There is no reason why minimising chip count or manufacturing cost could not be used.

To summarise the complete methodology that is proposed, and which has been demonstrated in the design of byte stream instruction sets for two very different languages is as follows:

(1) Design a *canonical instruction set.* If designing from scratch this is done on the basis of a one to one mapping from high level construct to low level instruction, within the bounds imposed by the target architecture. For a multiple language system a canonical instruction set is designed for each language and the instruction sets combined. Any semantically identical instructions are merged. This stage of the process is not automated and is where the designers' expertise must be brought to bear. If the methodology is used to refine an existing instruction set, then this is used as the canonical instruction set.

(2) Write compilers for each language under consideration and generate code using the canonical instruction set. Care should be taken to avoid the introduction of redundant information, for example by use of existing intermediate codes. A sample body of code should be obtained by compilation of programs to be used *within the target environment.* This body of code will be used to obtain statistical data to drive ISGEN.

(3) Specify design rules. These specify ways of deriving new instructions from existing instructions within the constraints imposed by the target architecture, with the aim of improving the instruction set with respect to the design criterion. It must be possible to quantify the degree of benefit obtainable by application of a rule in a particular case. This is the other stage in the process where the designers' expertise may be brought to bear.

(4) Derive a new instruction set by repeated application of the design rules, along the lines used by ISGEN, generating one instruction at a time. Initially and after each instruction is generated statistical information must be obtained for each instruction. This may be by substitution of each instruction and analysis of the resultant code, or may be by transformation of existing statistical data obtained initially. In the latter case the transformation effectively forms part of the design rule.

(5) Apply the new instruction set. This may be by simple peephole substitution of the canonical instruction set, or may be by modification of the existing compiler. If desired only some of the proposed instructions need be used, and the whole design process repeated with the new instruction set as the canonical instruction set. This may be the preferred route if the new instruction set has been used in a modified compiler rather than peephole substituted, in order to take advantage of global interactions of instructions.

The main problem with ISGEN is in the implementation of design rules. Each represents about 500-1000 lines of C, and needs a lot of care and attention. Each rule is very conscious of the data structures it is handling and the design criterion it is trying to meet. Whilst running ISGEN on a POLY instruction set to meet the criterion of compactness was straightforward, changing ISGEN to minimise memory-processor bandwidth for BCPL would be a major operation. Changing the system to handle RISC rather than byte stream instruction sets would probably involve starting from scratch. There is certainly little scope for making anything other than the minutest changes to the underlying algorithm. In an attempt to overcome some of these problems a language for specifying canonical instruction sets and design rules has been devised. This is the subject of the next chapter.

4. DL - A Design Language

The preceding chapter has shown that ISGEN is a cumbersome program to extend, and that implementing new design rules within it is far from trivial. Refinement of the selection algorithm is also prohibitively laborious. In large part this is because of the interdependency between the rules and the underlying algorithm. It would be of immense help if the rules and the canonical instruction set on which they are to operate could be specified in some independent fashion. To this end a design language, DL, has been designed. DL is implemented as a front end to the Unix C compiler. As it stands at present DL is geared towards byte stream instruction sets. However it is an advance on ISGEN, since it offers scope for use of arbitrary rules and saving functions.

4.1. Specification of DL

There are essentially four aspects to specifying a rule:

(1) *Matching* - A specification of what sequence of opcodes, or opcodes and arguments this rule will deal with. For example the rule that generates a new instruction with a smaller range for one argument is a rule that takes any opcode with one of its arguments.

(2) *Saving* - A function, which given a particular sequence of opcodes or opcodes and arguments, will return the benefit accrued were the rule to be applied to that sequence. Clearly this requires access to statistical data about the instruction set.

(3) *Specification* - A specification of the characteristics of an instruction derived by this rule in terms of the characteristics of the opcodes and arguments from which it is derived.

(4) *Generation* - A specification of how and under what circumstances the new instruction may replace existing instructions, and how to calculate the argument values for the new opcode if substituted.

To complete the system we need a specification of the characteristics of the canonical instruction set.

Fig. 1 shows an example DL program with just one rule, which specifies instructions with smaller argument ranges. This rule is more complex than that used in the ISGEN example of the previous chapter, in that the argument range may start from any value, not just zero. The rule is introduced by the word RULE (note that DL reserved words are in upper case, and variables in lower case) and terminated by a matching ENDRULE. The rule is given a name after RULE, purely for clarity. Following the RULE statement are a number of subsidiary information variables. These are used to hold subsidiary information calculated in the saving stage of the DL program that may be used by the specification and generation stages. In this case *range_start* specifies the start of the argument range and *size* the size of the argument for a particular application of the rule.

Matching is introduced by a MATCH clause. This specifies a sequence of opcodes, introduced by the keyword OPCODE. Each opcode may specify a number of subsidiary arguments with an ARG keyword and finally an ARGLIST for those arguments, such as switch tables whose size depends on another argument. The language at present assumes that there is only one arglist at most for any opcode, and that it follows any ordinary arguments. It is far from clear how to represent the size of an arglist, which is usually a simple function of one of the preceding

```
/* Example Design Language Program */

GET "iset.h"

RULE shrink_argument
INT range_start ;
INT size ;
/* Generate an instruction with a smaller argument */
  MATCH
   {
    OPCODE opc
    ARG    arg                          /* Must be more than 1 byte */
   }

  SAVING
   {
    INT best_start ;
    INT best_size ;
    INT best_saving ;
    best_saving = 0 ;
    /* Find the best size to shrink to. */
    FOR size ( 1 ) TO ( arg . SIZE - 1 )
     {
      INT range_freq ;
      INT range ;
      INT saving ;
      range = 1 << ( size * 8 );
      range_start = find_range ( arg, range ) ;
      range_freq = find_freq ( arg, range_start,
                               range_start + range - 1 ) ;
      saving = range_freq * ( arg . SIZE - size ) ;
      IF ( saving > best_saving )
       {
        best_start  = range_start ;
        best_size   = size ;
        best_saving = saving ;
       }
     }
    /* Now return the results */
    range_start = best_start ;
    size        = best_size ;
    RESULT best_saving ;
   }

  SPEC
   {
    OPCODE new_opc : "(" + opc . TEXT + size + "BYTESBASE" + range_start +
                arg . ID + ")",
                1, opc . NARGS
      ARGS new_arg : ( new_arg . ID == arg . ID ) -> size,
                     opc [ new_arg .ID ] . SIZE
   }

  GENERATION (( arg . VALUE >= range_start ) &&
         ( arg . VALUE <= ( range_start + ( 1 << ( size * 8 ))))))
   {
    OPCODE  new_opc
      ARGS  new_arg : opc [ new_arg . ID ] . VALUE
   }
ENDRULE  /* shrink_argument ( range_start, size ) */
```

Fig. 1. Example DL Program

arguments. The current implementation ignores this by not translating arglists. This was one of the problems that faced ISGEN, and must be added at some stage. However it is a major

complication. How for example are rules that transform an arglist into a number of fixed arguments to be handled? How is the combination of two opcodes with arglists to be dealt with? For the time being the issue is ignored in the interests of completing the design of a viable language. The DL evaluator may call the saving function for each possible opcode and argument sequence. The keywords OPCODE, ARG and ARGLIST each specify a name by which the relevant item may be addressed. In the example the match sequence is for a single opcode and argument. The generator will be invoked for each possible combination of an opcode and one of its arguments. Care must be exercised at this stage. It is clearly possible to specify a match for say a sequence of four opcodes, but with say 49 canonical instructions this could involve 49^4 (around 6 million) invocations on the first pass, and on pass 207 to find the last new opcode 255^4 (around 4 billion) invocations. More than 2 opcodes in a sequence is essentially to be discouraged. If larger numbers of opcodes are to be matched then consideration of the algorithm of the DL evaluator is required to prevent all possible matches being carried out.

The key part of the rule is the saving section, introduced by the SAVING keyword. This has a format rather like a C function, although much stripped down, and with a syntax that draws some ideas from BCPL. Variables may be specified. By implication the routine has available the opcodes and arguments specified in the MATCH section, and the variables specified by the RULE statement for returning subsidiary information. There are six types available in DL, INT's (integers), BOOL's (booleans), STRING's (text), OPCODE's (opcode data structure), ARG's (argument data structure) and ARGLIST's (arglist data structure). Fields within data structure types may be referenced using the "." operator. Possible fields are SIZE, ID and TEXT. In addition OPCODE types can reference their arguments using the notation:

opc[*argnumber*]

The saving function may be called for any opcode, arg, arglist combination possible within the MATCH specification, and returns as result the saving that would result from applying the rule to that combination. Subsidiary information is put in the subsidiary variables before a result is returned. Typically the DL evaluator keeps a record of the best saving possible, and the subsidiary information for the rule. Access to the statistical data is provided by external functions, which are in fact written in C. Two are used here, *find_range* and *find_freq*. It is not

difficult to add extra routines as needed for particular rules, although a complete system should have a wide range of useful routines built in. Using the saving function the DL evaluator decides which rule to apply to which opcode and arguments in order to create the maximum possible saving.

Having decided on which rule to apply to which opcodes and arguments the DL evaluator may need to specify new instructions. The specification section of a DL program is used to create a specification routine. This takes details of opcodes and arguments together with any subsidiary information as provided by the saving section and sets up data structures to describe the new instructions. The specification is introduced by the OPCODE keyword, and specifies a name by which the opcode may be referred; this is separated by a colon from initialisation data, giving the textual name of the new instruction, the opcode's size (which for 256 opcodes is typically one byte), and the number of arguments it has. These may be derived from existing data, as is the case here where the new opcode has the same number of arguments as the existing opcode. For each opcode there is an ARGS subsidiary specification. This gives a name to reference the argument, and initialisation data, being the new argument's size. This is invoked once for each argument in turn, with the argument ID field being set automatically; effectively an implied FOR loop. Finally there is an ARGLIST specification to add an arglist specification if necessary. This is not currently implemented.

The last part of the rule is the GENERATION section, which handles peephole substitution of instructions. Once again this is used to create a function. The peephole optimiser scans along the sample compiled code. Unless the sequence of opcodes and arguments being read meets the rule MATCH specification the code is copied unchanged. If its does match the generation routine is called. It tests the condition given after the GENERATION keyword, and if this succeeds does a peephole substitution. The generation routine returns TRUE or FALSE depending on whether or not the substitution was carried out. The GENERATION section specifies a sequence of opcodes to substitute. Each opcode is introduced by the OPCODE keyword, and followed by the name of an opcode. This inserts the ID field of that opcode in a number of bytes given by the SIZE field of the opcode. For each opcode there is a subsidiary ARGS field, which is used for each argument in the new opcode, and specifies a value to insert

for that argument. This is the only part of a DL program where the VALUE selector of an ARG may be used. In addition there is an ARGLIST specification to insert an ARGLIST if needed, although again this is not yet implemented.

In addition to specifying rules a DL program has to specify the canonical instruction set on which it is to operate. This is done by an initial specification section at the head of a DL program. This may be kept separate and inserted in the main DL program by means of a GET directive, as has been done with the example included above. Fig. 2 shows part of the specification of the canonical instruction set for BCPL described in section 3.2. It is similar to the SPEC section of a rule, but opcodes, args and arglists are not given names. Enough information is available here to enable the initial statistics to be gathered for use in instruction set generation.

From this description the reader should be able to understand a DL program. The language is implemented using the YACC parser generator under Unix, to produce a C program. This is then combined with an evaluator and library routines written in C. Library routines may be added as required by the instruction set designer, but do require an understanding of the underlying data structures in the evaluator. The complete YACC grammar for DL is given in Appendix B.

4.2. The implementation of DL

The translation of a DL program yields one main specification program and three routines for each rule. Details of instructions are held in a data structure similar to that used for ISGEN. The specification of ISGEN provides sufficient data to enable instruction frequency data, argument frequency data and instruction pair data to be obtained from a sample of compiled code. It is quite likely that more data might be needed within the specification section if DL is to be used on certain types of optimisation, for example details of bus loading by instructions. This can only be found out by experience. It is certainly adequate for work on dynamic code size, if sample data is presented as an instruction by instruction execution profile, since then the same statistics as used for static data size may be employed. A more general version of DL might provide opportunity for the specification of statistical data required, as well as instruction format. This is beyond the scope of this dissertation.

```
SPEC
 {
  OPCODE "unknown",      1, 0

  OPCODE "globalpush",    1, 1
    ARGS 4

  OPCODE "staticpush",    1, 1
    ARGS 4

  OPCODE "localpush",     1, 1
    ARGS 4

  OPCODE "push",        1, 0

  OPCODE "pushbyte",      1, 0

  OPCODE "globalpushleft", 1, 1
    ARGS 4

  OPCODE "staticpushleft", 1, 1
    ARGS 4

  OPCODE "localpushleft", 1, 1
    ARGS 4

  OPCODE "immediate",     1, 1
    ARGS 4

  OPCODE "globalpop",     1, 1
    ARGS 4

  OPCODE "staticpop",     1, 1
    ARGS 4

  OPCODE "localpop",      1, 1
    ARGS 4

  OPCODE "pop",         1, 0

  OPCODE "popbyte",       1, 0

  . . . . . .
  . . . . . .

  OPCODE "unlessuntil",   1, 1
    ARGS 4

  OPCODE "switch",       1, 2
    ARGS 4
 }
```

Fig. 2 Initial Specification of a DL Program

The algorithm implemented in the prototype DL compiler is essentially that employed by ISGEN. The saving possible by applying each design rule to each sequence of opcodes and arguments is evaluated, and the instruction giving the greatest saving selected. Peephole substitution is then carried out immediately (unlike ISGEN), and a new set of statistics obtained. The use of peephole substitution after each instruction will hopefully avoid some of the absurd instructions produced by the use of "derived" statistics in ISGEN.

DL is very much a prototype system, to demonstrate that in principle such a language is possible. The ability to separate design rules from the underlying generation algorithm is clearly vital if there is to be any scope for investigating other generation algorithms. Whatever its limitations, it is clear that DL is at least as powerful as ISGEN.

5. Theoretical Models

There have been a number of attempts over the years to place instruction set design on a sound theoretical footing. These have taken two forms. Modelling of the structure of instruction sets has been used to estimate the most compact instruction set that can be obtained to support a particular language [Flynn84]. The consideration of the entropy of the problem has been used to examine code size [Wade75] and redundancy in addressing [Hammerstrom77].

Both these approaches have been hampered by the need to constrain the problem in ways that are often unreasonable, and the need to make approximations of behaviour of instruction sets. The net effect is usually that predictions of behaviour are wildly over optimistic. Flynn suggests instruction sets that are ten times as compact as existing machines, something he cannot himself achieve in practice. His approach, with its basis in essentially *ad hoc* canonical interpretive measures is not well parameterised, and offers little hope of extension or improvement. It will not be discussed further, but must not be rejected out of hand. Although the theory of canonical interpretive measures is not particularly helpful, Flynn does raise many of the questions that need to be handled by a model of instruction set behaviour.

Analyses involving the entropy of an instruction stream offer far more hope. Wade and Stigall with their consideration of the theoretical basis of instruction set refinement give considerable theoretical backing to the approach used by ISGEN and DL. Hammerstrom and Davidson take a more sophisticated approach and analyse data on the dynamic behaviour of instruction set addressing. Their suggestion that memory references in the IBM S/370 contain up to 99% redundant information is worthy of serious consideration. Such a level of predictability would reduce memory processor bandwidth substantially, if only through the ability to design effective memory caches. If such an analysis could be extended to instruction stream and data stream in general the benefit would be enormous. These models and their limitation are considered in this chapter with regard to the canonical instruction set for BCPL presented in chapter three.

There is a need to investigate the algorithm used by ISGEN and DL, to ascertain whether they do lead to the best possible instruction set, and if not where the limitations lie. The analysis that they perform is based on statistical analysis of a large body of code. The size of code sample needed to obtain reliable results is an important consideration. Examining excessive amounts of code is time consuming; the minimum necessary for deriving the instruction set should be used.

From the design point of view it would be useful to have a model of behaviour that could predict the savings attainable by the use of automatic instruction set generators with far less computer time. This would then make experimentation with the canonical instruction set design more feasible. In the final section of this chapter the derivation of such a model is considered.

5.1. Entropy

The concepts of the information content and entropy of a stream of symbols and their mathematical analysis are essentially due to Shannon [Shannon48]. There are many textbooks on information theory that provide an introduction [for example Abramson63]. The approach has great promise as a mathematical formulation but as it stands there are two serious flaws, at least in Wade and Stigall's work [Wade75], discussed in section 1.7. The first is the treatment of an instruction stream as purely a sequence of opcodes. In reality opcodes have arguments, and this needs to be considered. The second failing is the assumption that an instruction stream can

be considered as a sequence of independent symbols. This is clearly absurd. Consider byte stream code compiled for the canonical BCPL instruction set described in chapter three. The probability of the occurrence of a CALL instruction is virtually zero after a GLOBALPOP instruction, but is very high after a GLOBALPUSH instruction. After a GLOBALPUSH 73 (global 73 being the BCPL routine WRITES) the probability approaches unity. A far better approach is to consider an instruction set as an m-stage Markov process, where the probability of an instruction occurring is a function of the previous m instructions in the stream. This is the approach used by Hammerstrom and Davidson [Hammerstrom77], who consider the sequence of addresses referenced in IBM S/370 code as an m-stage Markov process as m tends to ∞. If we consider the preceding m symbols, $(s_{j_1},s_{j_2}, \ldots, s_{j_m})$ as a single *state* then the conditional probability of the next symbol being s_i is $P(s_i \,/\, s_{j_1},s_{j_2}, \ldots, s_{j_m})$. By analogy the information obtained if s_i occurs whilst we are in state $(s_{j_1},s_{j_2}, \ldots, s_{j_m})$ is

$$I(s_i \,/\, s_{j_1},s_{j_2}, \ldots, s_{j_m}) = \log(\frac{1}{P(s_i \,/\, s_{j_1},s_{j_2}, \ldots, s_{j_m})}) \ .$$

Averaging over all possible states we can obtain a formula for the average information in the instruction stream [see Abramson63].

$$H = \sum_{S^{m+1}} P(s_{j_1},s_{j_2}, \ldots, s_{j_m},s_i) \ . \log \ (\frac{1}{P(s_i \,/\, s_{j_1},s_{j_2}, \ldots, s_{j_m})}) \ .$$

where the notation S^k, the k^{th} extension of S means all possible streams of symbols of length k in a stream of symbols S and $P(s_{j_1},s_{j_2}, \ldots, s_{j_m},s_i)$ is the probability of the state $(s_{j_1},s_{j_2}, \ldots, s_{j_m},s_i)$ occurring. This is essentially the approach which Hammerstrom and Davidson took to its limit.

To tabulate these results for various values of m is straightforward, if expensive in machine resources. Table 5.1 shows the results considering only the opcode bytes in 500K of canonical BCPL code (116125 opcodes) and the results considering all the bytes of the instruction stream for different numbers of stages in the Markov process. The figures for opcodes alone show the weakness of Wade's assumption that the opcodes are independent. We see that even as a first order Markov process the reduction in entropy is nearly 28%, more than one bit per symbol. For fifth order processes we are approaching a mere one bit requirement per symbol. When

Order	Entropy	
	Opcodes Only (bits/symbol)	Complete Instructions (bits/symbol)
0	3.98	3.24
1	2.87	2.64
2	2.35	2.28
3	2.03	2.01
4	1.62	1.77
5	1.19	1.26
6	0.80	1.03
7	0.51	0.97
8	0.32	0.81
9	0.20	0.50
10	0.13	
15	0.04	
20	0.02	
25	0.01	

Table 4.1 Opcode and Instruction Entropies as N-order Markov Processes

considering the entire instruction stream, we get far less order. This is not surprising since we are throwing away the information we have on instruction structure, with arguments following opcodes. This is without even imposing knowledge that arguments follow opcodes in a predefined way. Such small entropies lead to problems when considering the encoding of the instruction set. More than one writer has suggested the use of Huffman encoded instructions and bit addressed machines. Indeed such a machine, the Burroughs B1700, was built [Wilner72]. However Huffman encoding only offers the best approximation to the optimal encoding using a binary coding. With entropies around 0.01 bits per symbol almost all instructions will have less than one bit of information in them. Clearly a one bit encoding is inappropriate. The manner in which the instructions are to be represented will need much more

encoding. This is the problem that in practice faces Hammerstrom and Davidson.

Hammerstrom and Davidson also have the problem that their address data is calculated at run-time. The handling of dynamic data like this is a potential minefield, not one which they investigate.

There is considerable scope for developing these ideas, and they could form a dissertation in their own right. However further investigation is beyond the scope of this dissertation. It is sufficient to note the data as presented. From our point of view we may wish to note that the zero order model suggests less than four bits per symbol, and it may be sensible to consider nybble-stream[9] architectures.

5.2. The Theoretical Basis of ISGEN

The whole approach to the design of instruction sets presented here has been based on the principle of refinement of a canonical instruction set by addition of "special" instructions by peephole refinement. By this we mean the addition of extra instructions over the minimum necessary for a viable machine in order to handle particularly common cases more efficiently. The problem of selecting which "special" instructions to add to support particular constructs is equivalent to a standard problem in dynamic programming, the stock cutting problem.

Let us suppose that our instructions are partitioned into t types, T_1, T_2, ..., T_t. This may be at the level of partitioning into data access, data manipulation and flow of control instructions ($t = 3$), or it may be at the level of the canonical instruction set, with initially one instruction only of each type, and t possibly quite large. This latter partitioning is of greater relevance, since it reflects the behaviour of ISGEN in refining a canonical instruction set. We now construct our full instruction set by adding a number of special instructions to support each particular type. For each type T_i let there be x_i special instructions. In any body of compiled code we observe that

[9] Where a nybble is half a byte, i.e. four bits.

instructions of type T_i occur with frequency f_i. Clearly the higher this frequency the larger we might expect x_i to be, since then we increase the saving due to these instructions. For any given type adding special instructions will save space s_i, given by some function g_i of x_i

$$s_i = g_i(x_i)$$

And thus the total space saved will be

$$S = \sum_{i=1}^{t} s_i$$

$$= \sum_{i=1}^{t} g_i(x_i)$$

We wish to maximise S subject to the constraint $\sum_{1}^{t} x_i = c$, i.e. there is a limit on the number of special instructions permitted, typically 256 less the number of canonical instructions. In the case of instruction set design we know also that if we define

$$\Delta s_{i,j} = g_i(j) - g_i(j-1)$$

then $\Delta s_{i,j-1} \leq \Delta s_{i,j}$ for all $i \in 1, \ldots, t$ and $j > 1$

Under this constraint we have the stock cutting problem. It is a standard result that a "greedy" algorithm of adding one special instruction, the best available, at a time until we have reached the constraint, i.e. we have c special instructions will yield an optimal selection of instructions.

This immediately reveals the source of one of the problems found with ISGEN, for the stock cutting problem assumes that the values $\Delta s_{i,j}$ are independent. This is not always the case with ISGEN. The saving due to the creation of ((GLOBALPUSHBYTEARG1)ARG1=74) is very dependent on whether or not ((GLOBALPUSHBYTEARG1)CALL) has already been created. However errors such as this tend to come late on in the generation of an instruction set, and it is perhaps reasonable to believe that the errors due to this sort of interdependence are small, and that if not optimal, the instruction set generated by ISGEN or DL is not too sub-optimal. If we wish to generalise the problem, such that arbitrary dependencies are allowed, then we have a general problem which is believed to be NP-complete. However there are a series of approximations known to the general problem, which involve using lookahead when substituting. The greater the lookahead, the closer the approximation. This is a justification for incorporating lookahead into ISGEN or DL's algorithm. We should then have to consider the use of α–β

pruning techniques, common in computer chess programs, in order to reduce computation time. The current version of ISGEN takes 5 to 10 minutes with no lookahead.

The amount of statistical data used by ISGEN is large. For the BCPL instruction set of chapter three nearly half a megabyte of code was analysed, for POLY nearly a quarter of a megabyte. If this is to be refined and re-analysed for each instruction (as is the case with DL), then it is worth considering how much code needs to be analysed. Essentially we need to be sure that the $\Delta s_{i,j}$ giving the greatest saving in each case is correct. We have only an approximation to $\Delta s_{i,j}$, based on analysis of our data sample. ISGEN and DL would perhaps do well to incorporate a confidence test to check the reliability of $\Delta s_{i,j}$. For the early stages of design this could reduce substantially the degree of statistical analysis required, whilst instruction generation could be halted once no more reliable data remained. Under these conditions lookahead becomes much more viable.

This is fine as a methodology of instruction set design, apart from the fact that deriving an estimate of values of $\Delta s_{i,j}$ is extremely difficult, involving much computer time. This is in essence the weakness of ISGEN and DL. The amount of computer time required to generate an instruction set is a hindrance to experimentation with the canonical instruction set. It would be a considerable help with this aspect of instruction set design if we could derive a model that would enable us to predict the relative merits of different canonical instruction sets.

5.3. A Model of Instruction Set Design

The model considered here considers the case of optimising the space occupied by compiled code, but the technique is identical for any other quantifiable design criterion. Let us suppose we can approximate the benefit functions, g_i, by a decaying exponential.

Let s_i, x_i and f_i be defined as above. Let us suppose there exist non-negative constants a_i, such that:

$s_i = \sigma f_i (1-e^{-a_i x_i})$, where σ is a suitably dimensioned constant.

Thus the total space saved is:

$$S = \sigma \sum_{i=1}^{t} f_i (1 - e^{-a_i x_i}) \qquad \text{(Eqn. 1)}$$

subject to the constraint $\sum x_i = c$, where c is the number of special instructions we have room for in our instruction set of 256 instructions.The attraction of this model is that it is in some senses natural; negative exponential terms are what one might expect. If values of a_i cannot be found to model real choices of instructions accurately we may have to find a different version of Eqn. 1 or even reject the model completely.

We use the method of Lagrange Multipliers to maximise the saving. A proof derived by Smith in [Bennett87] is repeated in Appendix C. He shows that a choice of x_k given by

$$x_k = \frac{1}{ba_k} \log_e \left(\frac{\eta_k}{r}\right) \qquad \text{(Eqn. 2)}$$

will maximise S, where

$$\eta_k = (a_k f_k)^b$$

$$r = \prod_{i=1}^{t} \left[a_i f_i\right]^{d_i} \cdot e^{-ac}$$

$$b = \sum_{i=1}^{t} d_i$$

and $d_i = \prod_{\substack{j=1 \\ j \neq i}}^{t} a_j$.

The formula of Eqn. 2 as given is unsuitable for computation. With a_i around 5×10^{-2} and i around 10, d_i and b are of the order of 10^{-12}, making r and η_k of the form $10^{(10^{-11})}$, far too close to unity for easy manipulation by computer. This is solved by noting that the last term may be rewritten

$$\log_e \left(\frac{\eta_k}{r}\right) = \log_e (\eta_k) - \log_e (r)$$

$$\log_e (\eta_k) = \log_e (a_k f_k)^b$$

$$= b \cdot \log_e (a_k f_k)$$

$$\text{and } \log_e (r) = \log_e \left[\prod_{i=1}^{t} \left[a_i f_i \right]^{d_i} \cdot e^{-ac} \right]$$

$$= \sum_{i=1}^{t} d_i \log_e (a_i f_i) - ac$$

These formulas now involve only numbers of the order of 10^{-12}, well within the capabilities of an ordinary computer.

We tested the validity of this model by looking at the first of our examples, the design of an instruction set to support BCPL using ISGEN. We used the information on the benefit accruing as each instruction is added given by ISGEN to fit Eqn. 1 and obtain values for a_i and σf_i. For the purpose of our experiments we divided the canonical instructions into three types, and looked at the predicted and observed support for these three types in the final instruction set. The three groupings were data access instructions, data manipulation instructions and flow of control instructions. In the canonical instruction set we had 14, 21 and 14 instructions respectively supporting these three types. Compiled canonical code is observed to contain 72%, 3% and 25% of the three types respectively.

We tried fitting Eqn. 1 to our data, to obtain values of a_i, f_i and hence obtain predictions of the number of instructions of each type to be incorporated in the instruction set. There is a slight problem in that instructions formed by combination of instructions of two different types end up belonging to a new sort of combined type (for example `CALLGLOBAL`), but this problem is overcome by counting such an instruction as half an instruction of one type and half of the other. Statistically fitting Eqn. 1 by least squares regression is far from satisfactory, particularly since we have no model of error behaviour. However Table 5.2 shows the best results that we could achieve using our model, Eqn. 1. The prediction of number of opcodes is not too bad; ISGEN suggests a partition of 137, 10 and 60 is optimal. However the predicted saving of 72400 bytes is out by a factor of nearly 5 from the saving of 358762 bytes in the size of the code sample achieved with the instruction set selected by ISGEN.

The model of Eqn. 1. would appear inadequate for the task. It is unable to give really reliable

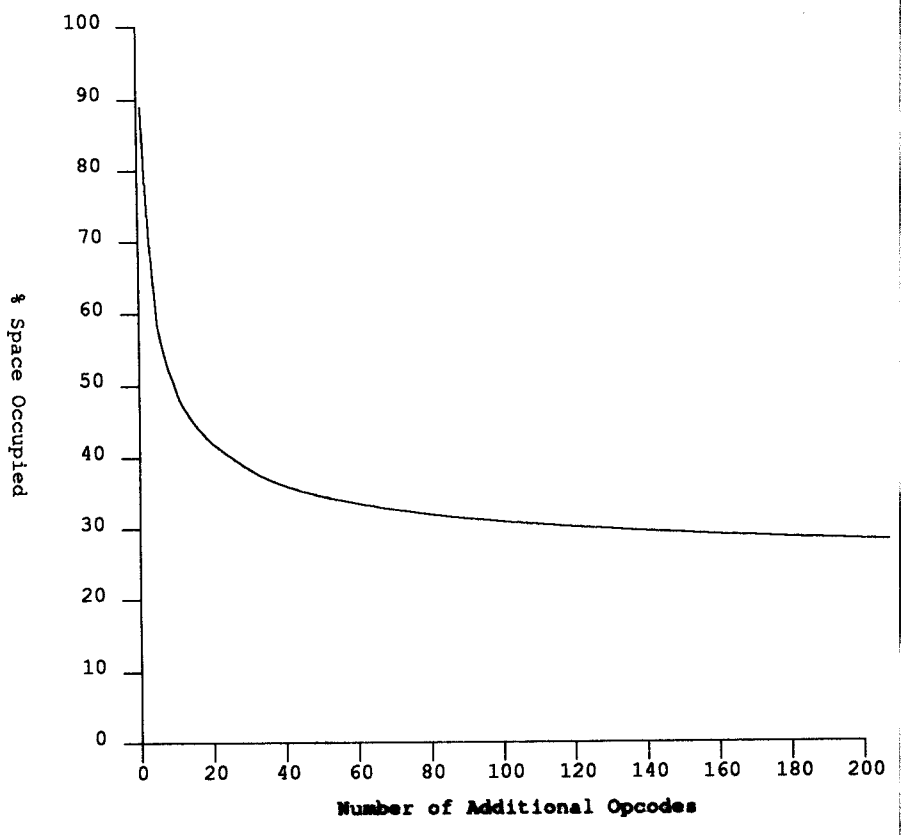
i	σf_i	a_i	x_i	s_i
1	133000	0.003	127	42100
2	12000	0.04	16	6550
3	54000	0.009	64	23700
Total	———	———	207	72400

Table 5.2 Model Instruction set distribution for three types

predictions of the number of opcodes of different types required. Its predictions of the savings to be achieved bear no discernible relationship to reality. To be of any use in practice we would not measure σf_i and a_i by curve fitting of completed data, but would attempt to estimate them in advance. This would of course lead to even less believable predictions.

The source of the problem with this model can be seen if we look at the savings that accrue as we add opcodes with ISGEN. A graph of space occupied by the sample compiled code as a percentage of its initial size as we incorporate in turn each additional opcode suggested by ISGEN is given in Graph 5.1. The fact is that whether we use 110 or 140 opcodes of type 1, or 10 or 20 of type 2 or 60 or 70 of type 3 makes very little difference. Almost all our benefit is coming from the first few special opcodes that we add. 90% of the space saved is achieved from the first 44 additional opcodes of all types. Thereafter additional opcodes make very little difference. When we look at our model we see that this is where we are on the flat part of the exponential decay. We can also explain the error in estimation of savings. This is due to error in fitting the first rapidly decaying part of the exponential. Slight errors in estimation of a_i will lead to gross under or over estimates of the savings to be expected. Our efforts to refine to the last opcode the addition of opcodes to our instruction set is really pointless. We can achieve 90% of our aim with an instruction set of just 93 opcodes. If we look at less easily quantified measures of benefit, such as dynamic code size or memory bus loading, we are unlikely to be able to quantify the benefit due to a particular opcode with as much as 10% accuracy anyway.

For comparison we took a second example, an instruction set for the programming language



Graph 5.1 - Percentage space saving due to additional opcodes

POLY. This is a far more sophisticated language than BCPL with a polymorphic type system. We might expect to need a larger instruction set, but even here 90% of the benefit possible with a full complement of 256 opcodes is achieved using just 116.

It seems there is a simple message for the designer of a byte stream architecture. Only add those extra opcodes you really need. There really is little point in filling up an instruction set with little used opcodes. Fewer opcodes mean less microcode, and more silicon for hardware assistance. It also makes the compiler writer's job easier, with fewer options in selecting opcodes. Under these conditions many of the benefits seen from RISC technology may be possible, whilst retaining the advantages found with byte stream architecture - the thesis proposed at the end of chapter one.

Conclusion

In this dissertation a methodology of instruction set design has been presented which I believe to be an advance on existing approaches. It offers the scope for accurate assessment of the quality of an instruction set and the automation of much of the drudgery involved in instruction set design. The computer designer has greater freedom to experiment with underlying principles, secure in the knowledge that he can always obtain the best possible results with his target architecture.

This work is not complete; it opens up new avenues of research. The instruction sets specified by programs such as ISGEN must still be implemented, itself a laborious process. There is no reason why ISGEN could not be made to generate microcode, or even perhaps silicon layouts.

ISGEN and DL clearly must be generalised to cope with designs using other than byte stream instruction sets. However this need not be restricted to the conventional software - hardware interface. The principles involved could as well be applied to the derivation of super combinators from combinatory logic as to design of computer instruction sets.

Analysis of the mechanism of instruction set design has forced a closer study of the theoretical basis of computer instruction sets. What little we have seen has revealed the need for further investigation in this area. This dissertation may help to promote such research.

Bibliography

This bibliography is in two parts, one for primary references, the other for secondary references. Primary references are works that are central to this thesis and which have played major parts in the evolution of the ideas contained herein. Secondary references are material that is needed for clarification and information on specific points, but which is not necessary for a general understanding of the subject.

Primary References

Chu75

Chu Y. ed., *High-level Language Computer Architecture.* Academic Press, 1975.

Colwell85

Colwell R.P. *et.al., Instruction Sets and Beyond: Computers Complexity and Controversy.* Computer, 18(9), 8-19 (September 1985).

Hammerstrom77

Hammerstrom D.W. and Davidson E.S., *Information Content of CPU Memory Referencing Behaviour.* Proceedings of the Fourth Annual ACM/IEEE Symposium on Computer Architecture, 184-192, March 1977.

Huffman52

Huffman D.A., *A Method for the Construction of Minimum-Redundancy Codes.* Proceedings of the IRE, 40(9), 1098-1101 (September 1952).

Johnsson82

Johnsson R.K. and Wick J.D., *An Overview of the MESA Processor Architecture.* Proceedings of the ACM Symposium on Architectural Support for Programming Languages and Operating Systems, 225-234, Palo Alto, March 1982.

Knuth71

Knuth D.E., *An Empirical Study of FORTRAN Programs.* Software - Practice and Experience, 1(2), 105-133 (1971).

Patterson80

Patterson D.A. and Ditzel D.R. *The Case for the Reduced Instruction Set Computer.* Computer Architecture News, 8(6), 25-33 (October 1980).

Patterson82

Patterson D.A. and Sequin C.H., *A VLSI RISC.* Computer, 15(9), 8-21 (September 1982).

Richards84

Richards M., *The Design of CINTCODE.* Personal Communication, 1984.

Shannon48

Shannon C.E., *A Mathematical Theory of Communication.* Bell Systems Technical Journal, 27, 379-423 and 623-656 (1948).

Siewiorek82

Siewiorek D.P., Bell C.G. and Newell A., *Computer Structures: Principles and Examples.* McGraw Hill, 1982.

Sweet82

Sweet R.E. and Sandman J.G., *Empirical Analysis of the MESA Instruction Set.* Proceedings of the ACM Symposium on Architectural Support for Programming Languages

and Operating Systems, 235-243, Palo Alto, March 1982.

Tanenbaum78

Tanenbaum A.S., *Implications of Structured Programming for Machine Architecture.* Communications of the ACM, 21(3), 237-246 (March 1978).

Wade75

Wade J.F. and Stigall P.D., *Instruction Design to Minimize Program Size.* Proceedings of the Second Annual ACM/IEEE Symposium on Computer Architecture, 41-44, 1975.

Weicker84

Weicker R.P., *Dhrystone: A Synthetic Systems Programming Benchmark.* Communications of the ACM, 27(10), 1013-1030 (October 1984).

Wilkes51

Wilkes M.V., *The Best Way to Design an Automatic Calculating Machine.* Manchester University Computer Inaugural Conference, Ferranti Ltd., London, July 1951.

Secondary References

Abramson63

Abramson N., *Information Theory and Coding.* McGraw Hill, 1963.

Anderson61

Anderson J.P., *A Computer for Direct Execution of Algorithmic Languages.* Proceedings of the AFIPS Fall Joint Computer Conference, 20, 184-193 (1961), *reported in* Chu75.

Baldwin84

Baldwin T.J., *An Implementation of the Tripos Operating System for the BBC micro.* Dissertation for the Computer Science Tripos, Cambridge University, May 1984.

Bashkov67

Bashkov T.R., Sasson A. and Kronfeld A., *System Design of a FORTRAN machine.* IEEE Transactions on Electronic Computers, EC-16(4), 485-499 (August 1967).

Bennett81

Bennett J.P., *A Comparison of FORTRAN compilers on the IBM S/370.* Contribution to

Seminar for the Computer Science Tripos, Cambridge University, 1981.

Bennett87

Bennett J.P. and Smith G.C., *The Need for Reduced Byte Stream Instruction Sets. To be published in* The Computer Journal, 1989.

Brooks83

Brooks P.T.M., *Quality of Code produced by the BCP Compiler.* Personal Communication, 1983.

Clark80

Clark T.J.W., Gladstone P.J.S., MacLean C.D. and Norman A.C., *SKIM - The S, K, I Reduction Machine.* Proceedings of the LISP Conference, 128-135, Stanford, August 1980.

Cownie86

The Meiko Computing Surface, an application of the INMOS Transputer. Seminar given at Bath University, England by J.H. Cownie, December 1986.

Davenport83

A Comparison of Franz LISP and PSL. Personal Communication, 1983.

DEC81

VAX Architecture Handbook. Digital Equipment Corporation, 1981.

Dewar83

Dewar R.B.K., McCann A.P., Jardine C. and McLaren N.M., *Macro-Spitbol on the IBM 370.* Cambridge University Computing Service, 1983.

Evans86

Evans R.D., *OCODE generated by the TBCPL compiler.* Personal communication, 1986.

Flynn84

Flynn M.J. and Hoevel L.W., *Measures of Ideal Execution Architectures.* IBM Journal of Research and Development, 28(4), 356-369 (July 1984).

Foderaro80

Foderaro J.K. and Sklower K., *Franz LISP Manual.* University of California, Berkeley, 1980.

Griss82

Griss M.L., Benson E. and Maguire G.Q. Jr., *PSL: A Portable LISP System.* Proceedings of the ACM Symposium on LISP and Functional Programming, 88-97, 1982.

Griswold71

Griswold R.E., Poage J.F. and Polonsky I.P., *The SNOBOL4 Programming Language, Second Edition.* Prentice Hall, 1971.

Grune79

Grune D., *Some Statistics on ALGOL 68 Programs.* SIGPLAN Notices, 14(7), 38-46 (July 1979).

HLH85

Microcoding the Orion. High Level Hardware, Oxford, 1985.

Hopper53

Hopper G.M. and Mauchly J.W., *Influence of Programming Techniques on the Design of Computers.* Proceedings of the IRE, 41(10), 1250-1254 (October 1953).

Lonergan61

Lonergan W. and King P., *Design of the B5000 System.* Datamation 7, 28-32 (1961), *reported in* Chu75.

Marks80

Marks B., *Compilation to Compact Code.* IBM Journal of Research and Development 24(6), 684-691 (November 1980).

Matthews85

Matthews D.C.J., *Poly Manual.* Cambridge University Computer Laboratory Technical Report No. 63 (1985).

Melbourn65

Melbourn A.J. and Pugmire J.M., *A Small Computer for Direct Processing of FORTRAN Statements.* The Computer Journal, 8, 24-27 (April 1965).

Motorola79

MC68000 User's Manual. Motorola, 1979.

National83

NS32032-6, NS32032-4 High-Performance Microprocessors. Preliminary Specification, National Semiconductor, 1983.

Needham82

Needham R.M. and Herbert A.J., *The Cambridge Distributed Computing System.* Addison Wesley, 1982.

Padget83

Padget J.A. and ffitch J.P., *Perq BCPL Documentation.* School of Mathematics, University of Bath, 1983.

3rivers81

PERQ System Software Reference Manual. Three Rivers Computer Corporation, July 1981.

Rice71

Rice R. and Smith W.R., *SYMBOL - A Major Departure from Classic Software Dominated von Neumann Computing Systems.* Proceedings of the AFIPS Spring Joint Computer Conference, 38, 575-587 (1971).

Rice81

Rice R., *The Chief Architect's Reflections on Symbol IIR.* Computer, 14(7), 49-54 (July 1981).

Richards79

Richards M. *et al., Tripos - A Portable Operating System for Mini-computers.* Software - Practice and Experience, 9(7), 513-526 (July 1979).

Richards80

Richards M. and Whitby-Strevens C., *BCPL - The Language and its Compiler.* Cambridge University Press, 1980.

Richards82

Richards J. and Jobson C., *BCPL for the BBC Microcomputer.* Acornsoft, 1983.

Rosen68

Rosen S., *Hardware Design Reflecting Software Requirements.* Proceedings of the AFIPS Fall Joint Computer Conference, 33, 1443-1449 (1968).

Shapiro72

Shapiro M.D., *A SNOBOL Machine: Functional Architectural Concepts of A String Processor.* Dissertation, Purdue University, Lafayette, Indianna, 1982, *reported in* Chu75.

Schulthess84

Schulthess P.U., *A Reduced High-Level-Language Instruction Set.* IEEE Micro, 4(3), 55-66 (June 1984).

Tafvelin75

Tafvelin S. and Wikstrom A. *Aspects of Compact Programs and Directly Executed Languages.* BIT 15(2), 203-214 (1975).

Tanenbaum84

Tanenbaum A.S., *Structured Computer Organisation, Second Edition.* Prentice Hall, 1984.

Weber67

Weber H., *A Microprogrammed Implementation of EULER on IBM System 360 Model 30.* Communications of the ACM, 10(9) 549-558 (September 1967).

Wilner72

Wilner W.T., *Design of the Burroughs B1700.* Proceedings of the AFIPS Fall Joint Computer Conference, 41, 489-497, (1972).

Wilson83

Wilson A.R. *The Design of the NS16032.* Personal Communication, 1983.

Wilson85

A System for Preloading Tripos commands. Systems Research Group Note, Cambridge University Computer Laboratory, 1985.

Wilson86

The Acorn RISC Machine. Seminar given at Bath University, England by A.R. Wilson and D. Flynn, December 1986.

Wirth86

Wirth N., *Microprocessor Architectures: A Comparison Based on Code Generation by Compiler.* Communications of the ACM, 29(10), 978-990 (October 1986).

This is another line

I hope we are now complete - discard this page.

Appendix A. ISGEN Output

This shows sample runs from ISGEN in generating a BCPL instruction set (section 3.4) and in generating a POLY instruction set (section 3.6). For clarity a certain amount of debugging information has been removed in each case.

A.1 A BCPL Instruction Set

```
ISGEN version 1.30
 49 redn 89.04%, total 89.04%, (IMMEDIATEBYTEARG1)
 50 redn 88.04%, total 78.39%, (LOCALPUSHBYTEARG1)
 51 redn 89.69%, total 70.31%, (GLOBALPUSHBYTEARG1)
 52 redn 91.54%, total 64.36%, (CALLBYTEARG1)
 53 redn 90.88%, total 58.49%, (LOCALPOPBYTEARG1)
 54 redn 95.61%, total 55.92%, (IFNILBYTEARG1)
 55 redn 96.74%, total 54.10%, (GLOBALPOPBYTEARG1)
 56 redn 96.70%, total 52.31%, (JUMPBYTEARG1)
 57 redn 97.27%, total 50.88%, ((GLOBALPUSHBYTEARG1)(CALLBYTEARG1))
 58 redn 97.19%, total 49.46%, ((IMMEDIATEBYTEARG1)ARG1=0)
 59 redn 97.25%, total 48.10%, (STATICPUSHBYTEARG1)
 60 redn 98.02%, total 47.14%, (STATICPUSHLEFTHALFWORDARG1)
 61 redn 98.19%, total 46.29%, (IMMEDIATEARG1=4294967295)
 62 redn 98.34%, total 45.52%, ((LOCALPUSHBYTEARG1)ARG1=0)
 63 redn 98.50%, total 44.84%, (GLOBALPUSHHALFWORDARG1)
 64 redn 98.60%, total 44.21%, ((IMMEDIATEBYTEARG1)ARG1=1)
 65 redn 98.68%, total 43.63%, (PLEASEPUSH)
 66 redn 98.70%, total 43.06%, (UNLESSUNTILBYTEARG1)
 67 redn 98.74%, total 42.52%, ((LOCALPOPBYTEARG1)(LOCALPUSHBYTEARG1))
 68 redn 98.86%, total 42.03%, (ENDFORARG2=1)
```

```
 69 redn 99.04%, total 41.63%, ((LOCALPUSHBYTEARG1)ARG1=1)
 70 redn 99.05%, total 41.23%, (FORBYTEARG1)
 71 redn 99.04%, total 40.83%, ((FORBYTEARG1)BYTEARG2)
 72 redn 99.10%, total 40.47%, (EQ(IFWHILEBYTEARG1))
 73 redn 99.10%, total 40.10%, ((ENDFORARG2=1)BYTEARG1)
 74 redn 99.10%, total 39.74%, (((FORBYTEARG1)BYTEARG2)BYTEARG3)
 75 redn 99.15%, total 39.41%, ((STATICPUSHBYTEARG1)(CALLBYTEARG1))
 76 redn 99.16%, total 39.08%, (((ENDFORARG2=1)BYTEARG1)BYTEARG3)
 77 redn 99.17%, total 38.75%, (IMMEDIATEHALFWORDARG1)
 78 redn 99.18%, total 38.43%, ((LOCALPUSHBYTEARG1)ARG1=2)
 79 redn 99.18%, total 38.12%, ((STATICPUSHLEFTHALFWORDARG1)BYTEARG1)
 80 redn 99.17%, total 37.80%, (JUMPHALFWORDARG1)
 81 redn 99.27%, total 37.53%, (REPEATBYTEARG1)
 82 redn 99.33%, total 37.28%, (LOCALPUSHLEFTBYTEARG1)
 83 redn 99.35%, total 37.03%, (PLUSPOP)
 84 redn 99.38%, total 36.81%, ((LOCALPOPBYTEARG1)ARG1=0)
 85 redn 99.39%, total 36.58%, ((IMMEDIATEBYTEARG1)ARG1=2)
 86 redn 99.42%, total 36.37%, ((LOCALPUSHBYTEARG1)ARG1=3)
 87 redn 99.47%, total 36.18%, ((LOCALPOPBYTEARG1)ARG1=1)
 88 redn 99.50%, total 36.00%, ((GLOBALPUSHHALFWORDARG1)(CALLBYTEARG1))
 89 redn 99.52%, total 35.82%, (((GLOBALPUSHBYTEARG1)(CALLBYTEARG1))ARG2=0)
 90 redn 99.54%, total 35.66%, ((LOCALPUSHBYTEARG1)ARG1=4)
 91 redn 99.54%, total 35.50%, ((LOCALPOPBYTEARG1)ARG1=2)
 92 redn 99.58%, total 35.34%, ((GLOBALPUSHBYTEARG1)((IMMEDIATEBYTEARG1)ARG1=0))
 93 redn 99.59%, total 35.20%, ((IMMEDIATEBYTEARG1)ARG1=3)
 94 redn 99.61%, total 35.06%, (SWITCHBYTEARG2)
 95 redn 99.62%, total 34.93%, ((LOCALPUSHBYTEARG1)ARG1=5)
 96 redn 99.62%, total 34.79%, (((GLOBALPUSHBYTEARG1)(CALLBYTEARG1))ARG2=1)
 97 redn 99.62%, total 34.66%, (((GLOBALPUSHBYTEARG1)(CALLBYTEARG1))ARG2=2)
 98 redn 99.63%, total 34.53%, ((GLOBALPUSHBYTEARG1)(IMMEDIATEBYTEARG1))
 99 redn 99.63%, total 34.40%, (((GLOBALPUSHBYTEARG1)(CALLBYTEARG1))ARG2=3)
100 redn 99.64%, total 34.28%, ((LOCALPOPBYTEARG1)ARG1=3)
101 redn 99.67%, total 34.17%, ((LOCALPUSHBYTEARG1)((IMMEDIATEBYTEARG1)ARG1=0))
102 redn 99.68%, total 34.06%, (EQ(UNLESSUNTILBYTEARG1))
103 redn 99.70%, total 33.96%, ((LOCALPOPBYTEARG1)ARG1=4)
104 redn 99.71%, total 33.86%, ((IMMEDIATEBYTEARG1)ARG1=10)
105 redn 99.71%, total 33.76%, ((LOCALPUSHBYTEARG1)ARG1=6)
106 redn 99.71%, total 33.66%, (((LOCALPOPBYTEARG1)(LOCALPUSHBYTEARG1))ARG2=1)
107 redn 99.72%, total 33.57%, (((IMMEDIATEBYTEARG1)ARG1=0)(EQ(IFWHILEBYTEARG1)))
108 redn 99.73%, total 33.47%, ((SWITCHBYTEARG2)HALFWORDARG1)
109 redn 99.73%, total 33.38%, ((IMMEDIATEBYTEARG1)ARG1=4)
110 redn 99.73%, total 33.29%, (((GLOBALPUSHBYTEARG1)(CALLBYTEARG1))ARG2=4)
111 redn 99.73%, total 33.21%, (EQLOGOR)
112 redn 99.74%, total 33.12%, (NE(IFWHILEBYTEARG1))
113 redn 99.74%, total 33.03%, (LOGAND(IFWHILEBYTEARG1))
114 redn 99.74%, total 32.95%, (STATICPUSHHALFWORDARG1)
115 redn 99.75%, total 32.86%, (REPEATHALFWORDARG1)
116 redn 99.75%, total 32.78%, ((GLOBALPUSHBYTEARG1)(GLOBALPUSHBYTEARG1))
117 redn 99.75%, total 32.70%, (((LOCALPUSHBYTEARG1)ARG1=0)((IMMEDIATEBYTEARG1)ARG1=0))
118 redn 99.76%, total 32.62%, (((LOCALPOPBYTEARG1)(LOCALPUSHBYTEARG1))ARG2=2)
119 redn 99.76%, total 32.55%, (((GLOBALPUSHBYTEARG1)(CALLBYTEARG1))ARG2=5)
120 redn 99.77%, total 32.47%, ((LOCALPOPBYTEARG1)ARG1=5)
121 redn 99.77%, total 32.40%, ((IMMEDIATEBYTEARG1)(EQ(IFWHILEBYTEARG1)))
122 redn 99.78%, total 32.33%, (((IMMEDIATEBYTEARG1)ARG1=0)(GLOBALPOPBYTEARG1))
123 redn 99.78%, total 32.25%, ((JUMPBYTEARG1)RETURN)
124 redn 99.79%, total 32.19%, (GLOBALPOPHALFWORDARG1)
125 redn 99.79%, total 32.12%, (((IMMEDIATEBYTEARG1)ARG1=0)(PLUSPUSH))
126 redn 99.79%, total 32.05%, ((LOCALPUSHBYTEARG1)(IMMEDIATEBYTEARG1))
127 redn 99.80%, total 31.99%, ((STATICPUSHBYTEARG1)GOTO)
128 redn 99.80%, total 31.92%, ((GLOBALPUSHBYTEARG1)(PLUSPUSH))
129 redn 99.81%, total 31.86%, ((GLOBALPUSHBYTEARG1)((IMMEDIATEBYTEARG1)ARG1=1))
130 redn 99.81%, total 31.80%, (STATICPOPBYTEARG1)
131 redn 99.81%, total 31.74%, (((LOCALPOPBYTEARG1)(LOCALPUSHBYTEARG1))ARG1=0)
132 redn 99.82%, total 31.69%, (((LOCALPUSHBYTEARG1)ARG1=0)(IMMEDIATEBYTEARG1))
133 redn 99.82%, total 31.63%, ((LOCALPOPBYTEARG1)ARG1=6)
134 redn 99.82%, total 31.57%, (((GLOBALPUSHBYTEARG1)(CALLBYTEARG1))ARG2=6)
135 redn 99.83%, total 31.52%, (IFWHILEHALFWORDARG1)
136 redn 99.83%, total 31.47%, ((LOCALPUSHBYTEARG1)ARG1=8)
137 redn 99.83%, total 31.41%, ((LOCALPUSHBYTEARG1)ARG1=7)
138 redn 99.83%, total 31.36%, ((GLOBALPOPBYTEARG1)((IMMEDIATEBYTEARG1)ARG1=0))
139 redn 99.84%, total 31.31%, (((LOCALPOPBYTEARG1)(LOCALPUSHBYTEARG1))ARG1=1)
140 redn 99.84%, total 31.26%, (GR(IFWHILEBYTEARG1))
141 redn 99.84%, total 31.21%, ((GLOBALPUSHBYTEARG1)((GLOBALPUSHBYTEARG1)(CALLBYTEARG1)))
142 redn 99.85%, total 31.16%, ((IMMEDIATEBYTEARG1)(PLUSPUSH))
143 redn 99.85%, total 31.12%, ((JUMPBYTEARG1)((IMMEDIATEBYTEARG1)ARG1=0))
144 redn 99.85%, total 31.07%, ((IMMEDIATEBYTEARG1)(GLOBALPOPBYTEARG1))
145 redn 99.85%, total 31.02%, (((LOCALPUSHBYTEARG1)ARG1=0)(PLUSPUSH))
146 redn 99.85%, total 30.98%, (((IMMEDIATEBYTEARG1)ARG1=1)(EQ(IFWHILEBYTEARG1)))
```

```
147 redn 99.85%, total 30.93%, (PLUS(GLOBALPOPBYTEARG1))
148 redn 99.85%, total 30.89%, ((LOCALPUSHBYTEARG1)(PLUSPUSH))
149 redn 99.85%, total 30.84%, (GLOBALPUSHLEFTBYTEARG1)
150 redn 99.86%, total 30.80%, (((LOCALPOPBYTEARG1)(LOCALPUSHBYTEARG1))ARG1=2)
151 redn 99.87%, total 30.76%, (((STATICPUSHBYTEARG1)(CALLBYTEARG1))ARG2=0)
152 redn 99.87%, total 30.72%, (((STATICPUSHLEFTHALFWORDARG1)BYTEARG1)(GLOBALPUSHBYTEARG1))
153 redn 99.87%, total 30.68%, ((LOCALPOPBYTEARG1)(((FORBYTEARG1)BYTEARG2)BYTEARG3))
154 redn 99.87%, total 30.64%, ((LOCALPOPBYTEARG1)((IMMEDIATEBYTEARG1)ARG1=0))
155 redn 99.87%, total 30.60%, ((IMMEDIATEBYTEARG1)ARG1=5)
156 redn 99.87%, total 30.56%, (((GLOBALPUSHBYTEARG1)(CALLBYTEARG1))RETURN)
157 redn 99.88%, total 30.52%, ((IMMEDIATEBYTEARG1)ARG1=8)
158 redn 99.88%, total 30.49%, ((LOCALPOPBYTEARG1)((LOCALPUSHBYTEARG1)ARG1=0))
159 redn 99.88%, total 30.45%, ((IMMEDIATEBYTEARG1)ARG1=7)
160 redn 99.88%, total 30.41%, (((LOCALPUSHBYTEARG1)ARG1=1)((IMMEDIATEBYTEARG1)ARG1=0))
161 redn 99.88%, total 30.38%, ((GLOBALPUSHBYTEARG1)(GLOBALPOPBYTEARG1))
162 redn 99.88%, total 30.34%, (((IMMEDIATEBYTEARG1)ARG1=1)(PLUSPUSH))
163 redn 99.88%, total 30.31%, (((GLOBALPUSHBYTEARG1)(CALLBYTEARG1))(JUMPBYTEARG1))
164 redn 99.88%, total 30.27%, ((LOCALPUSHBYTEARG1)ARG1=9)
165 redn 99.89%, total 30.24%, ((LOCALPUSHBYTEARG1)ARG1=11)
166 redn 99.89%, total 30.20%, (((IMMEDIATEBYTEARG1)ARG1=1)(GLOBALPOPBYTEARG1))
167 redn 99.89%, total 30.17%, ((IMMEDIATEBYTEARG1)ARG1=6)
168 redn 99.89%, total 30.14%, ((GLOBALPUSHBYTEARG1)PLUS)
169 redn 99.89%, total 30.10%, (ENDFORBYTEARG1)
170 redn 99.89%, total 30.07%, (LS(IFWHILEBYTEARG1))
171 redn 99.89%, total 30.04%, (((IMMEDIATEBYTEARG1)ARG1=0)EQ)
172 redn 99.89%, total 30.01%, (((STATICPUSHBYTEARG1)(CALLBYTEARG1))ARG1=0)
173 redn 99.89%, total 29.97%, ((IMMEDIATEBYTEARG1)ARG1=32)
174 redn 99.89%, total 29.94%, ((GLOBALPUSHBYTEARG1)(IFWHILEBYTEARG1))
175 redn 99.89%, total 29.91%, (UNLESSUNTILHALFWORDARG1)
176 redn 99.89%, total 29.88%, (((LOCALPOPBYTEARG1)(LOCALPUSHBYTEARG1))ARG1=3)
177 redn 99.89%, total 29.84%, (REPEATUNTILBYTEARG1)
178 redn 99.89%, total 29.81%, ((IMMEDIATEBYTEARG1)ARG1=20)
179 redn 99.89%, total 29.78%, (((STATICPUSHLEFTHALFWORDARG1)BYTEARG1)((GLOBALPUSHBYTEARG1)
(CALLBYTEARG1)))
180 redn 99.90%, total 29.75%, ((STATICPUSHHALFWORDARG1)(CALLBYTEARG1))
181 redn 99.90%, total 29.72%, ((EQLOGOR)(IFWHILEBYTEARG1))
182 redn 99.90%, total 29.69%, (((IMMEDIATEBYTEARG1)ARG1=0)PLUS)
183 redn 99.90%, total 29.66%, ((ENDFORBYTEARG1)BYTEARG3)
184 redn 99.90%, total 29.63%, ((LOCALPOPBYTEARG1)ARG1=7)
185 redn 99.90%, total 29.60%, (((LOCALPUSHBYTEARG1)ARG1=0)((IMMEDIATEBYTEARG1)ARG1=1))
186 redn 99.91%, total 29.57%, (((GLOBALPUSHHALFWORDARG1)(CALLBYTEARG1))ARG1=256)
187 redn 99.91%, total 29.55%, (((STATICPUSHBYTEARG1)(CALLBYTEARG1))ARG2=2)
188 redn 99.91%, total 29.52%, (((STATICPUSHBYTEARG1)(CALLBYTEARG1))ARG2=1)
189 redn 99.91%, total 29.49%, (((STATICPUSHBYTEARG1)(CALLBYTEARG1))ARG2=3)
190 redn 99.91%, total 29.46%, (((IMMEDIATEBYTEARG1)ARG1=0)MINUS)
191 redn 99.91%, total 29.44%, (((GLOBALPUSHBYTEARG1)(CALLBYTEARG1))ARG2=7)
192 redn 99.91%, total 29.41%, (((LOCALPOPBYTEARG1)(LOCALPUSHBYTEARG1))ARG1=4)
193 redn 99.91%, total 29.39%, (LELOGAND)
194 redn 99.91%, total 29.36%, (((SWITCHBYTEARG2)HALFWORDARG1)BYTEARG1)
195 redn 99.91%, total 29.33%, ((LOCALPUSHBYTEARG1)((IMMEDIATEBYTEARG1)ARG1=1))
196 redn 99.91%, total 29.31%, (((STATICPUSHLEFTHALFWORDARG1)BYTEARG1)(JUMPBYTEARG1))
197 redn 99.92%, total 29.28%, ((GLOBALPUSHHALFWORDARG1)ARG1=256)
198 redn 99.92%, total 29.26%, (((LOCALPUSHBYTEARG1)ARG1=0)PLUS)
199 redn 99.92%, total 29.23%, ((IMMEDIATEBYTEARG1)ARG1=48)
200 redn 99.92%, total 29.21%, ((GLOBALPUSHBYTEARG1)(PLUSPOP))
201 redn 99.92%, total 29.18%, (((LOCALPUSHBYTEARG1)ARG1=1)(PLUSPUSH))
202 redn 99.92%, total 29.16%, (((LOCALPUSHBYTEARG1)ARG1=2)((IMMEDIATEBYTEARG1)ARG1=0))
203 redn 99.92%, total 29.14%, (((FORBYTEARG1)BYTEARG2)HALFWORDARG3)
204 redn 99.92%, total 29.11%, (((GLOBALPUSHBYTEARG1)(CALLBYTEARG1))ARG2=9)
205 redn 99.92%, total 29.09%, ((IMMEDIATEBYTEARG1)ARG1=9)
206 redn 99.92%, total 29.07%, ((LOCALPUSHBYTEARG1)PLUS)
207 redn 99.92%, total 29.04%, ((LOCALPOPBYTEARG1)ARG1=8)
208 redn 99.92%, total 29.02%, (((GLOBALPUSHHALFWORDARG1)(CALLBYTEARG1))ARG1=259)
209 redn 99.92%, total 29.00%, (((IMMEDIATEBYTEARG1)ARG1=0)(JUMPBYTEARG1))
210 redn 99.93%, total 28.98%, (((LOCALPOPBYTEARG1)(LOCALPUSHBYTEARG1))ARG2=3)
211 redn 99.93%, total 28.96%, ((GLOBALPOPBYTEARG1)RETURN)
212 redn 99.93%, total 28.94%, ((LOCALPUSHBYTEARG1)ARG1=13)
213 redn 99.93%, total 28.91%, (((ENDFORARG2=1)BYTEARG1)HALFWORDARG3)
214 redn 99.93%, total 28.89%, (LOGOR(IFWHILEBYTEARG1))
215 redn 99.93%, total 28.87%, (((IMMEDIATEBYTEARG1)ARG1=0)(PLUSPOP))
216 redn 99.93%, total 28.85%, (((GLOBALPUSHBYTEARG1)(CALLBYTEARG1))ARG2=10)
217 redn 99.93%, total 28.83%, (((GLOBALPUSHBYTEARG1)(CALLBYTEARG1))ARG2=8)
218 redn 99.93%, total 28.81%, ((GLOBALPUSHBYTEARG1)((IMMEDIATEBYTEARG1)ARG1=2))
219 redn 99.93%, total 28.79%, (REPEATWHILEBYTEARG1)
220 redn 99.93%, total 28.77%, ((LOCALPUSHBYTEARG1)ARG1=10)
221 redn 99.93%, total 28.75%, ((JUMPBYTEARG1)((STATICPUSHLEFTHALFWORDARG1)BYTEARG1))
222 redn 99.93%, total 28.73%, (((LOCALPOPBYTEARG1)ARG1=0)(((FORBYTEARG1)BYTEARG2)BYTEARG3))
223 redn 99.93%, total 28.71%, ((GLOBALPUSHHALFWORDARG1)ARG1=259)
```

```
224 redn 99.93%, total 28.70%, (((ENDFORBYTEARG1)BYTEARG3)ARG2=4294967295)
225 redn 99.93%, total 28.68%, (((LOCALPUSHBYTEARG1)ARG1=2)(PLUSPUSH))
226 redn 99.93%, total 28.66%, (((IMMEDIATEBYTEARG1)ARG1=0)(EQ(UNLESSUNTILBYTEARG1)))
227 redn 99.93%, total 28.64%, (((IMMEDIATEBYTEARG1)ARG1=1)EQ)
228 redn 99.93%, total 28.62%, ((GLOBALPUSHBYTEARG1)ARG1=74)
229 redn 99.93%, total 28.60%, (GE(IFWHILEBYTEARG1))
230 redn 99.93%, total 28.58%, ((JUMPHALFWORDARG1)RETURN)
231 redn 99.93%, total 28.56%, (((STATICPUSHBYTEARG1)(CALLBYTEARG1))ARG2=4)
232 redn 99.93%, total 28.54%, (((GLOBALPUSHHALFWORDARG1)(CALLBYTEARG1))ARG2=0)
233 redn 99.94%, total 28.53%, (((GLOBALPUSHBYTEARG1)(CALLBYTEARG1))ARG2=12)
234 redn 99.94%, total 28.51%, ((LOCALPUSHBYTEARG1)ARG1=12)
235 redn 99.94%, total 28.49%, ((LOCALPOPBYTEARG1)ARG1=9)
236 redn 99.94%, total 28.47%, ((GLOBALPOPBYTEARG1)ARG1=10)
237 redn 99.94%, total 28.45%, ((GLOBALPUSHBYTEARG1)(IMMEDIATEARG1=4294967295))
238 redn 99.94%, total 28.44%, ((JUMPBYTEARG1)((LOCALPUSHBYTEARG1)ARG1=0))
239 redn 99.94%, total 28.42%, (POPBYTE(((ENDFORARG2=1)BYTEARG1)BYTEARG3))
240 redn 99.94%, total 28.40%, (((LOCALPOPBYTEARG1)(LOCALPUSHBYTEARG1))ARG2=4)
241 redn 99.94%, total 28.38%, (((IMMEDIATEBYTEARG1)ARG1=1)PLUS)
242 redn 99.94%, total 28.37%, (((LOCALPUSHBYTEARG1)ARG1=0)(PLUSPOP))
243 redn 99.94%, total 28.35%, (NELOGAND)
244 redn 99.94%, total 28.33%, ((GLOBALPUSHBYTEARG1)ARG1=73)
245 redn 99.94%, total 28.32%, ((IMMEDIATEBYTEARG1)EQ)
246 redn 99.94%, total 28.30%, (((STATICPUSHBYTEARG1)(CALLBYTEARG1))ARG2=5)
247 redn 99.94%, total 28.28%, (LE(LOGAND(IFWHILEBYTEARG1)))
248 redn 99.94%, total 28.27%, (((IMMEDIATEBYTEARG1)ARG1=2)(EQ(IFWHILEBYTEARG1)))
249 redn 99.94%, total 28.25%, (((LOCALPOPBYTEARG1)ARG1=1)(((FORBYTEARG1)BYTEARG2)BYTEARG3))
250 redn 99.94%, total 28.23%, ((LOCALPUSHLEFTBYTEARG1)ARG1=1)
251 redn 99.94%, total 28.22%, (MINUS(GLOBALPOPBYTEARG1))
252 redn 99.94%, total 28.20%, ((JUMPBYTEARG1)(LOCALPUSHBYTEARG1))
253 redn 99.94%, total 28.19%, (((IMMEDIATEBYTEARG1)ARG1=0)(NE(IFWHILEBYTEARG1)))
254 redn 99.94%, total 28.17%, ((((LOCALPOPBYTEARG1)(LOCALPUSHBYTEARG1))ARG2=1)ARG1=0)
255 redn 99.94%, total 28.16%, ((IMMEDIATEARG1=4294967295)(EQ(IFWHILEBYTEARG1)))
```

A.2 A POLY Instruction Set

```
ISGEN version 1.30
 26 redn 93.05%, total 93.05%, (localBYTEARG1)
 27 redn 96.64%, total 89.92%, (returnARG1=0)
 28 redn 96.71%, total 86.96%, (indirectBYTEARG1)
 29 redn 96.72%, total 84.11%, (move_to_vecBYTEARG1)
 30 redn 96.87%, total 81.48%, (const_intBYTEARG1)
 31 redn 97.03%, total 79.06%, (io_vec_entrycall_closure)
 32 redn 97.15%, total 76.81%, (non_localARG2=1)
 33 redn 97.34%, total 74.76%, ((localBYTEARG1)(indirectBYTEARG1))
 34 redn 97.57%, total 72.94%, (reset_rARG1=1)
 35 redn 97.64%, total 71.22%, (const_addrBYTEARG1)
 36 redn 98.20%, total 69.94%, (returnARG1=1)
 37 redn 98.25%, total 68.72%, (resetARG1=1)
 38 redn 98.22%, total 67.49%, ((localBYTEARG1)ARG1=1)
 39 redn 98.33%, total 66.37%, ((non_localARG2=1)BYTEARG1)
 40 redn 98.42%, total 65.32%, (((non_localARG2=1)BYTEARG1)BYTEARG3)
 41 redn 98.57%, total 64.39%, ((const_intBYTEARG1)(io_vec_entrycall_closure))
 42 redn 98.68%, total 63.53%, (reset_rARG1=2)
 43 redn 98.71%, total 62.71%, ((localBYTEARG1)ARG1=0)
 44 redn 98.79%, total 61.95%, (jump_falseBYTEARG1)
 45 redn 98.84%, total 61.23%, ((localBYTEARG1)ARG1=2)
 46 redn 98.96%, total 60.60%, (get_storeARG1=2)
 47 redn 98.96%, total 59.96%, (((localBYTEARG1)(indirectBYTEARG1))ARG2=0)
 48 redn 98.95%, total 59.34%, (((localBYTEARG1)(indirectBYTEARG1))ARG2=1)
 49 redn 98.95%, total 58.71%, (non_localARG2=2)
 50 redn 98.94%, total 58.09%, ((move_to_vecBYTEARG1)ARG1=0)
 51 redn 98.94%, total 57.47%, ((move_to_vecBYTEARG1)ARG1=1)
 52 redn 98.98%, total 56.89%, (jumpBYTEARG1)
 53 redn 99.03%, total 56.34%, ((localBYTEARG1)ARG1=3)
 54 redn 99.04%, total 55.80%, ((((non_localARG2=1)BYTEARG1)BYTEARG3)(indirectBYTEARG1))
 55 redn 99.22%, total 55.36%, ((const_intBYTEARG1)ARG1=0)
 56 redn 99.33%, total 54.99%, ((localBYTEARG1)ARG1=4)
 57 redn 99.40%, total 54.66%, (returnARG1=2)
 58 redn 99.41%, total 54.34%, ((const_addrBYTEARG1)(move_to_vecBYTEARG1))
 59 redn 99.42%, total 54.03%, (get_storeBYTEARG1)
 60 redn 99.42%, total 53.72%, ((non_localARG2=2)BYTEARG1)
```

```
 61 redn 99.43%, total 53.41%, ((io_vec_entrycall_closure)ARG1=7)
 62 redn 99.44%, total 53.11%, ((const_addrBYTEARG1)(returnARG1=0))
 63 redn 99.45%, total 52.82%, ((const_intBYTEARG1)ARG1=1)
 64 redn 99.45%, total 52.53%, (((non_localARG2=2)BYTEARG1)BYTEARG3)
 65 redn 99.49%, total 52.26%, ((localBYTEARG1)ARG1=5)
 66 redn 99.50%, total 52.00%, (call_slARG3=0)
 67 redn 99.50%, total 51.74%, (reset_rARG1=3)
 68 redn 99.51%, total 51.49%, ((io_vec_entrycall_closure)ARG1=229)
 69 redn 99.53%, total 51.24%, ((move_to_vecBYTEARG1)ARG1=2)
 70 redn 99.53%, total 51.00%, (io_vec_entryraise_ex)
 71 redn 99.53%, total 50.76%, ((const_addrBYTEARG1)call_closure)
 72 redn 99.53%, total 50.52%, (((const_intBYTEARG1)(io_vec_entrycall_closure))ARG1=0)
 73 redn 99.56%, total 50.30%, ((localBYTEARG1)ARG1=6)
 74 redn 99.60%, total 50.09%, (((((non_localARG2=1)BYTEARG1)BYTEARG3)(indirectBYTEARG1))ARG3=1)
 75 redn 99.60%, total 49.89%, (call_closure(returnARG1=0))
 76 redn 99.61%, total 49.69%, (((localBYTEARG1)(indirectBYTEARG1))ARG1=1)
 77 redn 99.61%, total 49.50%, ((((non_localARG2=1)BYTEARG1)BYTEARG3)ARG3=1)
 78 redn 99.63%, total 49.32%, (call_closure(resetARG1=1))
 79 redn 99.64%, total 49.14%, ((localBYTEARG1)ARG1=7)
 80 redn 99.64%, total 48.97%, (call_closure(jump_falseBYTEARG1))
 81 redn 99.65%, total 48.80%, (call_closure(reset_rARG1=1))
 82 redn 99.66%, total 48.63%, ((const_intBYTEARG1)ARG1=48)
 83 redn 99.67%, total 48.47%, (const_addr(returnARG1=0))
 84 redn 99.67%, total 48.31%, (((const_intBYTEARG1)(io_vec_entrycall_closure))ARG1=1)
 85 redn 99.67%, total 48.15%, ((((localBYTEARG1)(indirectBYTEARG1))ARG2=0)ARG1=1)
 86 redn 99.68%, total 48.00%, ((((localBYTEARG1)(indirectBYTEARG1))ARG2=1)ARG1=1)
 87 redn 99.68%, total 47.84%, ((move_to_vecBYTEARG1)ARG1=3)
 88 redn 99.69%, total 47.69%, ((((non_localARG2=2)BYTEARG1)BYTEARG3)(indirectBYTEARG1))
 89 redn 99.70%, total 47.55%, ((localBYTEARG1)ARG1=8)
 90 redn 99.72%, total 47.42%, (((localBYTEARG1)(indirectBYTEARG1))ARG2=2)
 91 redn 99.72%, total 47.28%, (((((non_localARG2=1)BYTEARG1)BYTEARG3)(indirectBYTEARG1))ARG3=2)
 92 redn 99.72%, total 47.15%, ((call_slARG3=0)BYTEARG2)
 93 redn 99.73%, total 47.03%, ((io_vec_entrycall_closure)(returnARG1=0))
 94 redn 99.73%, total 46.90%, ((const_addrBYTEARG1)((move_to_vecBYTEARG1)ARG1=0))
 95 redn 99.73%, total 46.78%, ((const_addrBYTEARG1)((move_to_vecBYTEARG1)ARG1=1))
 96 redn 99.73%, total 46.65%, ((((non_localARG2=1)BYTEARG1)BYTEARG3)ARG3=2)
 97 redn 99.74%, total 46.53%, ((const_addrBYTEARG1)(returnARG1=1))
 98 redn 99.74%, total 46.41%, (const_addr(move_to_vecBYTEARG1))
 99 redn 99.74%, total 46.29%, (returnBYTEARG1)
100 redn 99.75%, total 46.17%, ((const_intBYTEARG1)ARG1=49)
101 redn 99.75%, total 46.06%, (del_handler_rBYTEARG1)
102 redn 99.75%, total 45.94%, ((io_vec_entrycall_closure)(resetARG1=1))
103 redn 99.76%, total 45.83%, ((get_storeBYTEARG1)ARG1=3)
104 redn 99.76%, total 45.72%, ((io_vec_entrycall_closure)(jump_falseBYTEARG1))
105 redn 99.76%, total 45.61%, (((const_intBYTEARG1)(io_vec_entrycall_closure))ARG2=7)
106 redn 99.76%, total 45.50%, ((reset_rARG1=1)(jump_falseBYTEARG1))
107 redn 99.77%, total 45.40%, ((((localBYTEARG1)(indirectBYTEARG1))ARG2=0)ARG1=0)
108 redn 99.77%, total 45.29%, ((((localBYTEARG1)(indirectBYTEARG1))ARG2=1)ARG1=0)
109 redn 99.77%, total 45.19%, (resetBYTEARG1)
110 redn 99.77%, total 45.08%, (((localBYTEARG1)ARG1=1)(move_to_vecBYTEARG1))
111 redn 99.78%, total 44.98%, (((localBYTEARG1)(indirectBYTEARG1))ARG1=0)
112 redn 99.78%, total 44.88%, ((localBYTEARG1)ARG1=9)
113 redn 99.79%, total 44.79%, (set_stack_valBYTEARG1)
114 redn 99.79%, total 44.69%, ((((localBYTEARG1)(indirectBYTEARG1))ARG2=0)ARG1=2)
115 redn 99.79%, total 44.60%, ((((localBYTEARG1)(indirectBYTEARG1))ARG2=1)ARG1=2)
116 redn 99.79%, total 44.51%, ((io_vec_entrycall_closure)(reset_rARG1=1))
117 redn 99.80%, total 44.42%, (((const_intBYTEARG1)(io_vec_entrycall_closure))ARG2=229)
118 redn 99.80%, total 44.33%, (((localBYTEARG1)(indirectBYTEARG1))ARG1=2)
119 redn 99.81%, total 44.25%, (non_localBYTEARG1)
120 redn 99.81%, total 44.16%, ((non_localBYTEARG1)BYTEARG2)
121 redn 99.81%, total 44.08%, (call_closure(returnARG1=1))
122 redn 99.82%, total 44.00%, (call_closure(reset_rARG1=2))
123 redn 99.82%, total 43.92%, ((non_localBYTEARG1)BYTEARG2)BYTEARG3)
124 redn 99.82%, total 43.85%, (set_handlerBYTEARG1)
125 redn 99.83%, total 43.77%, ((((non_localARG2=1)BYTEARG1)BYTEARG3)(indirectBYTEARG1))ARG3=3)
126 redn 99.83%, total 43.70%, ((move_to_vecBYTEARG1)ARG1=4)
127 redn 99.83%, total 43.62%, ((localBYTEARG1)ARG1=10)
128 redn 99.83%, total 43.55%, ((((non_localARG2=1)BYTEARG1)BYTEARG3)ARG3=3)
129 redn 99.84%, total 43.48%, ((((localBYTEARG1)(indirectBYTEARG1))ARG2=0)ARG1=3)
130 redn 99.84%, total 43.41%, ((((localBYTEARG1)(indirectBYTEARG1))ARG2=1)ARG1=3)
131 redn 99.84%, total 43.34%, ((get_storeBYTEARG1)ARG1=4)
132 redn 99.84%, total 43.27%, ((indirectBYTEARG1)ARG1=0)
133 redn 99.84%, total 43.20%, ((indirectBYTEARG1)ARG1=1)
134 redn 99.84%, total 43.13%, (((localBYTEARG1)(indirectBYTEARG1))ARG1=3)
135 redn 99.84%, total 43.06%, (reset_rBYTEARG1)
136 redn 99.84%, total 42.99%, (const_addr((move_to_vecBYTEARG1)ARG1=0))
137 redn 99.84%, total 42.92%, (const_addr((move_to_vecBYTEARG1)ARG1=1))
138 redn 99.85%, total 42.86%, (call_slBYTEARG3)
```

```
139 redn 99.85%, total 42.79%, (const_addr(returnARG1=1))
140 redn 99.85%, total 42.73%, ((call_slBYTEARG3)BYTEARG2)
141 redn 99.85%, total 42.67%, ((((((non_localARG2=1)BYTEARG1)BYTEARG3)(indirectBYTEARG1))
ARG3=1)ARG4=0)
142 redn 99.85%, total 42.60%, ((((((non_localARG2=1)BYTEARG1)BYTEARG3)(indirectBYTEARG1))
ARG3=1)ARG4=1)
143 redn 99.86%, total 42.54%, (((localBYTEARG1)ARG1=0)(move_to_vecBYTEARG1))
144 redn 99.86%, total 42.48%, (((const_addrBYTEARG1)(move_to_vecBYTEARG1))ARG2=2)
145 redn 99.86%, total 42.42%, ((returnBYTEARG1)ARG1=3)
146 redn 99.86%, total 42.36%, (((localBYTEARG1)ARG1=1)((move_to_vecBYTEARG1)ARG1=0))
147 redn 99.86%, total 42.30%, (((localBYTEARG1)ARG1=1)((move_to_vecBYTEARG1)ARG1=1))
148 redn 99.86%, total 42.25%, ((const_intBYTEARG1)ARG1=3)
149 redn 99.87%, total 42.19%, (((const_intBYTEARG1)(io_vec_entrycall_closure))ARG1=48)
150 redn 99.87%, total 42.13%, ((const_intBYTEARG1)ARG1=2)
151 redn 99.87%, total 42.08%, (((((non_localARG2=2)BYTEARG1)BYTEARG3)(indirectBYTEARG1))ARG3=1)
152 redn 99.87%, total 42.02%, (((call_slARG3=0)BYTEARG2)BYTEARG1)
153 redn 99.87%, total 41.97%, (((localBYTEARG1)ARG1=2)(move_to_vecBYTEARG1))
154 redn 99.87%, total 41.92%, ((((non_localARG2=2)BYTEARG1)BYTEARG3)ARG3=1)
155 redn 99.88%, total 41.86%, ((resetBYTEARG1)ARG1=2)
156 redn 99.88%, total 41.81%, ((io_vec_entrycall_closure)(returnARG1=1))
157 redn 99.88%, total 41.76%, ((const_addrBYTEARG1)raise_ex)
158 redn 99.88%, total 41.71%, ((reset_rARG1=2)(jump_falseBYTEARG1))
159 redn 99.88%, total 41.66%, ((((non_localARG2=1)BYTEARG1)BYTEARG3)(indirectBYTEARG1))ARG3=4)
160 redn 99.88%, total 41.61%, ((io_vec_entryraise_ex)ARG1=7)
161 redn 99.88%, total 41.56%, ((((const_intBYTEARG1)(io_vec_entrycall_closure))ARG1=0)ARG2=7)
162 redn 99.88%, total 41.51%, (const_addrcall_closure)
163 redn 99.89%, total 41.46%, ((((localBYTEARG1)(indirectBYTEARG1))ARG2=0)ARG1=4)
164 redn 99.89%, total 41.42%, ((((localBYTEARG1)(indirectBYTEARG1))ARG2=1)ARG1=4)
165 redn 99.89%, total 41.37%, ((localBYTEARG1)ARG1=11)
166 redn 99.89%, total 41.32%, (((non_localARG2=1)BYTEARG1)BYTEARG3)ARG3=4)
167 redn 99.89%, total 41.27%, ((move_to_vecBYTEARG1)ARG1=5)
168 redn 99.89%, total 41.23%, ((localBYTEARG1)(indirectBYTEARG1))ARG1=4)
169 redn 99.89%, total 41.18%, ((reset_rARG1=1)(returnARG1=0))
170 redn 99.89%, total 41.14%, ((((localBYTEARG1)(indirectBYTEARG1))ARG1=1)ARG2=2)
171 redn 99.90%, total 41.10%, ((io_vec_entrycall_closure)(reset_rARG1=2))
172 redn 99.90%, total 41.05%, (((io_vec_entrycall_closure)ARG1=7)(returnARG1=0))
173 redn 99.90%, total 41.01%, (((non_localARG2=1)BYTEARG1)ARG3=65535)
174 redn 99.90%, total 40.97%, ((resetBYTEARG1)ARG1=3)
175 redn 99.90%, total 40.93%, ((const_intBYTEARG1)ARG1=4)
176 redn 99.90%, total 40.89%, ((io_vec_entryraise_ex)ARG1=229)
177 redn 99.90%, total 40.85%, ((((const_intBYTEARG1)(io_vec_entrycall_closure))ARG1=0)ARG2=229)
178 redn 99.90%, total 40.81%, ((((((non_localARG2=1)BYTEARG1)BYTEARG3)(indirectBYTEARG1))
ARG3=2)ARG4=1)
179 redn 99.90%, total 40.77%, ((((((non_localARG2=1)BYTEARG1)BYTEARG3)(indirectBYTEARG1))
ARG3=2)ARG4=0)
180 redn 99.90%, total 40.73%, (((localBYTEARG1)ARG1=1)((localBYTEARG1)ARG1=1))
181 redn 99.90%, total 40.69%, (((const_intBYTEARG1)(io_vec_entrycall_closure))ARG1=49)
182 redn 99.90%, total 40.65%, (((const_addrBYTEARG1)(move_to_vecBYTEARG1))ARG2=3)
183 redn 99.90%, total 40.61%, ((((non_localBYTEARG1)BYTEARG2)BYTEARG3)ARG2=4)
184 redn 99.90%, total 40.57%, ((((non_localBYTEARG1)BYTEARG2)BYTEARG3)ARG2=3)
185 redn 99.90%, total 40.53%, (((localBYTEARG1)ARG1=0)((move_to_vecBYTEARG1)ARG1=0))
186 redn 99.90%, total 40.49%, (((localBYTEARG1)ARG1=0)((move_to_vecBYTEARG1)ARG1=1))
187 redn 99.91%, total 40.45%, (((io_vec_entrycall_closure)ARG1=7)(resetARG1=1))
188 redn 99.91%, total 40.42%, (((localBYTEARG1)ARG1=3)(move_to_vecBYTEARG1))
189 redn 99.91%, total 40.38%, (((io_vec_entrycall_closure)ARG1=7)(jump_falseBYTEARG1))
190 redn 99.91%, total 40.34%, (((((non_localARG2=2)BYTEARG1)BYTEARG3)(indirectBYTEARG1))ARG3=2)
191 redn 99.91%, total 40.31%, ((set_stack_valBYTEARG1)(resetARG1=1))
192 redn 99.91%, total 40.27%, ((((non_localARG2=1)BYTEARG1)BYTEARG3)(indirectBYTEARG1))ARG3=5)
193 redn 99.91%, total 40.24%, (((io_vec_entrycall_closure)ARG1=229)(returnARG1=0))
194 redn 99.91%, total 40.20%, ((((non_localARG2=2)BYTEARG1)BYTEARG3)ARG3=2)
195 redn 99.91%, total 40.17%, ((const_addrBYTEARG1)(io_vec_entrycall_closure))
196 redn 99.92%, total 40.13%, ((((localBYTEARG1)(indirectBYTEARG1))ARG2=0)ARG1=5)
197 redn 99.92%, total 40.10%, ((((localBYTEARG1)(indirectBYTEARG1))ARG2=1)ARG1=5)
198 redn 99.91%, total 40.07%, (((localBYTEARG1)ARG1=2)((move_to_vecBYTEARG1)ARG1=0))
199 redn 99.91%, total 40.03%, (((localBYTEARG1)ARG1=2)((move_to_vecBYTEARG1)ARG1=1))
200 redn 99.92%, total 40.00%, ((reset_rARG1=1)(jumpBYTEARG1))
201 redn 99.92%, total 39.96%, (((localBYTEARG1)ARG1=1)call_closure)
202 redn 99.92%, total 39.93%, ((((const_intBYTEARG1)(io_vec_entrycall_closure))ARG1=1)ARG2=7)
203 redn 99.92%, total 39.90%, ((returnBYTEARG1)ARG1=4)
204 redn 99.92%, total 39.87%, (((const_addrBYTEARG1)call_closure)(returnARG1=0))
205 redn 99.92%, total 39.83%, ((const_intBYTEARG1)ARG1=10)
206 redn 99.92%, total 39.80%, (((localBYTEARG1)(indirectBYTEARG1))ARG1=5)
207 redn 99.92%, total 39.77%, ((((non_localARG2=1)BYTEARG1)BYTEARG3)ARG3=5)
208 redn 99.92%, total 39.73%, (const_addr((move_to_vecBYTEARG1)ARG1=2))
209 redn 99.92%, total 39.70%, (((localBYTEARG1)ARG1=1)(io_vec_entrycall_closure))
210 redn 99.92%, total 39.67%, (((localBYTEARG1)ARG1=1)(returnARG1=0))
211 redn 99.92%, total 39.64%, (((const_intBYTEARG1)(io_vec_entrycall_closure))(returnARG1=0))
212 redn 99.92%, total 39.61%, ((move_to_vecBYTEARG1)ARG1=6)
```

```
213 redn 99.92%, total 39.57%, ((const_addrBYTEARG1)(returnARG1=2))
214 redn 99.92%, total 39.54%, (((io_vec_entrycall_closure)ARG1=229)(resetARG1=1))
215 redn 99.92%, total 39.51%, ((get_storeBYTEARG1)ARG1=5)
216 redn 99.92%, total 39.48%, (const_intARG1=300)
217 redn 99.92%, total 39.45%, (((((non_localARG2=1)BYTEARG1)BYTEARG3)ARG3=1)ARG1=3)
218 redn 99.92%, total 39.42%, ((reset_rARG1=1)(del_handler_rBYTEARG1))
219 redn 99.92%, total 39.39%, (((io_vec_entrycall_closure)ARG1=229)(jump_falseBYTEARG1))
220 redn 99.92%, total 39.36%, (((const_addrBYTEARG1)call_closure)(resetARG1=1))
221 redn 99.93%, total 39.33%, ((((localBYTEARG1)(indirectBYTEARG1))ARG2=2)ARG1=0)
222 redn 99.93%, total 39.30%, ((reset_rBYTEARG1)ARG1=4)
223 redn 99.93%, total 39.28%, ((((localBYTEARG1)(indirectBYTEARG1))ARG2=0)ARG1=6)
224 redn 99.93%, total 39.25%, (((const_intBYTEARG1)(io_vec_entrycall_closure))(resetARG1=1))
225 redn 99.93%, total 39.22%, (((const_addrBYTEARG1)call_closure)(jump_falseBYTEARG1))
226 redn 99.93%, total 39.19%, (((localBYTEARG1)(indirectBYTEARG1))ARG1=6)
227 redn 99.93%, total 39.16%, (((non_localARG2=1)BYTEARG1)ARG3=65531)
228 redn 99.93%, total 39.14%, ((((localBYTEARG1)(indirectBYTEARG1))ARG2=1)ARG1=6)
229 redn 99.93%, total 39.11%, (const_addrraise_ex)
230 redn 99.93%, total 39.08%, (((const_intBYTEARG1)(io_vec_entrycall_closure))(jump_falseBYTEAR$
231 redn 99.93%, total 39.05%, ((((const_intBYTEARG1)(io_vec_entrycall_closure))ARG1=1)ARG2=229)
232 redn 99.93%, total 39.03%, (((localBYTEARG1)ARG1=1)((move_to_vecBYTEARG1)ARG1=2))
233 redn 99.93%, total 39.00%, (((call_slBYTEARG3)BYTEARG2)BYTEARG1)
234 redn 99.93%, total 38.97%, ((move_to_vecBYTEARG1)ARG1=7)
235 redn 99.93%, total 38.95%, ((((localBYTEARG1)(indirectBYTEARG1))ARG2=2)ARG1=2)
236 redn 99.93%, total 38.92%, ((const_intBYTEARG1)ARG1=5)
237 redn 99.93%, total 38.89%, (((localBYTEARG1)ARG1=3)((move_to_vecBYTEARG1)ARG1=0))
238 redn 99.93%, total 38.87%, (((localBYTEARG1)ARG1=3)((move_to_vecBYTEARG1)ARG1=1))
239 redn 99.93%, total 38.84%, (((const_addrBYTEARG1)(move_to_vecBYTEARG1))ARG2=4)
240 redn 99.93%, total 38.82%, (((localBYTEARG1)ARG1=1)(reset_rARG1=1))
241 redn 99.93%, total 38.79%, (((localBYTEARG1)ARG1=1)((localBYTEARG1)ARG1=0))
242 redn 99.93%, total 38.76%, ((call_closure(reset_rARG1=1))(jump_falseBYTEARG1))
243 redn 99.94%, total 38.74%, (((((non_localARG2=1)BYTEARG1)BYTEARG3)ARG3=1)ARG1=4)
244 redn 99.94%, total 38.72%, ((localBYTEARG1)ARG1=12)
245 redn 99.94%, total 38.69%, (((((non_localARG2=1)BYTEARG1)BYTEARG3)ARG3=1)ARG1=5)
246 redn 99.94%, total 38.67%, ((((((non_localARG2=1)BYTEARG1)BYTEARG3)(indirectBYTEARG1))
ARG3=3)ARG4=1)
247 redn 99.94%, total 38.64%, ((set_stack_valBYTEARG1)(jumpBYTEARG1))
248 redn 99.94%, total 38.62%, (((io_vec_entrycall_closure)ARG1=7)(reset_rARG1=1))
249 redn 99.94%, total 38.60%, ((localBYTEARG1)ARG1=13)
250 redn 99.94%, total 38.57%, ((((localBYTEARG1)(indirectBYTEARG1))ARG2=0)ARG1=7)
251 redn 99.94%, total 38.55%, ((((localBYTEARG1)(indirectBYTEARG1))ARG2=1)ARG1=7)
252 redn 99.94%, total 38.53%, ((get_storeBYTEARG1)ARG1=6)
253 redn 99.94%, total 38.51%, ((((((non_localARG2=1)BYTEARG1)BYTEARG3)(indirectBYTEARG1))
ARG3=3)ARG4=0)
254 redn 99.94%, total 38.48%, (((localBYTEARG1)ARG1=1)((localBYTEARG1)ARG1=2))
255 redn 99.94%, total 38.46%, (((localBYTEARG1)ARG1=0)call_closure)
```

Appendix B. DL Grammar

This is the YACC parser for DL. A certain amount of header and trailer code is present, providing the interface to the rest of the compiler.

```
/*****************************************************************************
*****************************************************************************

              DDDDDD   LL       CCCCC
              DDDDDDD  LL      CCCCCCC
              DD   DD LL     CC
              DD   DD LL     CC
              DD   DD LL     CC
              DD   DD LL     CC
              DDDDDDD  LLLLLLLL  CCCCCCC
              DDDDDD   LLLLLLLL  CCCCC

      PPPPPPP   AAAA   RRRRRRR   SSSSS  EEEEEEEE RRRRRRR
      PPPPPPPP  AAAAAA  RRRRRRRR SSSSSSS EEEEEEEE RRRRRRRR
      PP  PP AA  AA RR  RR SS     EE     RR  RR
      PPPPPPP  AAAAAAAA RRRRRRRR SSSSSS  EEEEEE  RRRRRRRR
      PP     AA  AA RRRRRRR      SS EE     RRRRRRR
      PP     AA  AA RR RR      SS EE     RR RR
      PP     AA  AA RR  RR  SSSSSSSS EEEEEEEE RR  RR
      PP     AA  AA RR  RR  SSSSSS  EEEEEEEE RR  RR

*****************************************************************************
*****************************************************************************

          DLC - A Design Language Compiler
```

```
  This is the parser for the compiler. It is written for use by the YACC
  compiler compiler.

*****************************************************************************
*****************************************************************************/

/* First tokens that are in fact characters */

%token T_CURLYBRA  123          /* '{' */
%token T_CURLYKET  125          /* '}' */
%token T_COLON     58          /* ':' */
%token T_COMMA     44          /* ',' */
%token T_SEMICOLON  59          /* ';' */
%token T_POINT     46          /* '.' */
%token T_PLUS      43          /* '+' */
%token T_MUL       42          /* '*' */
%token T_REM       37          /* '%' */
%token T_BRA       40          /* '(' */
%token T_KET       41          /* ')' */
%token T_SQBRA      91          /* '[' */
%token T_SQKET      93          /* ']' */
%token T_DIV       47          /* '/' */
%token T_ASSIGN     61          /* '=' */
%token T_MINUS      45          /* '-' */
%token T_LT        60          /* '<' */
%token T_GT        62          /* '>' */
%token T_NOT       33          /* '!' */

/* The endmarker */

%token T_ENDMARKER  0

/* Now composite symbols */

%token T_EQ       257
%token T_COND      258
%token T_LE       259
%token T_LSHIFT    260
%token T_GE       261
%token T_RSHIFT    262
%token T_NE       263
%token T_LOGAND    264
%token T_LOGOR     265

/* Other data items */

%token T_NUMBER    266
%token T_NAME      267
%token T_STRLITERAL 268

/* Reserved words */

%token T_SPEC      269
%token T_OPCODE    270
%token T_ARGS      271
%token T_ARGLIST    272
%token T_RULE      273
%token T_ENDRULE    274
%token T_INT       275
%token T_STRING    276
%token T_BOOL      277
%token T_MATCH     278
%token T_ARG       279
%token T_SAVING    280
%token T_TEXT      281
%token T_SIZE      282
%token T_ID        283
%token T_VALUE     284
%token T_MARGS     285
%token T_TRUE      286
%token T_FALSE     287
%token T_RESULT    288
%token T_IF        289
%token T_TEST      290
%token T_ELSE      291
%token T_FOR       292
%token T_TO        293
%token T_DO        294
```

```
%token T_WHILE     295
%token T_GENERATION 296

/* Internal tokens */

%token T_UNKNOWN   297
%token T_GET       298
%token T_UMINUS    299          /* Used for precedence */

/* The start token */

%start  program

/* Operator precedence */

%left ','
%left T_LOGAND, T_LOGOR
%left T_EQ, T_NE, '<', T_LE, '>', T_GE
%left '+' '-'
%left '*' '/' '%'
%left T_LSHIFT T_RSHIFT
%left '!'
%left T_UMINUS

/* Code to include as part of the header */

%{

#include <stdio.h>
#include "dlc.h"
#include "cg.h"


/* Routines local to this section */

int             mknode () ;
int             mkname () ;
int             mknum () ;
int             mkstr () ;

%}

%%

/* The rules section */

program         : spec_part rules
                {

                 /* At this stage we should have an appropriate tree.
                   It ends up in yyval. */

                 $$ = mknode ( N_PROGRAM, $1, $2 ) ;
                }

              ; /* End of rule for <program> */

spec_part       : T_SPEC spec_block
                {

                 /* We needn't do anything here */

                 $$ = $2 ;
                }

              ; /* End of rule for <spec_sequence> */

spec_block      : '{' spec_list '}'
                {

                 /* We needn't do anything here */

                 $$ = $2 ;
                }

              ; /* End of rule for <spec_block> */

spec_list       : spec
```

```
                {
                 /* List of one. */
                 $$ = mknode ( N_SPEC_LIST, $1, NIL ) ;
                }
              | spec spec_list
                {
                 /* Build up the list */
                 $$ = mknode ( N_SPEC_LIST, $1, $2 ) ;
                }
              ; /* End of rule for <spec_list> */
spec          : opcode_spec
                {
                 /* Put in null elements for args and arglist */
                 $$ = mknode ( N_SPEC, $1, NIL, NIL ) ;
                }
              | opcode_spec args_spec
                {
                 /* Put in null elements for arglist */
                 $$ = mknode ( N_SPEC, $1, $2, NIL ) ;
                }
              | opcode_spec       arglist_spec
                {
                 /* Put in null elements for args */
                 $$ = mknode ( N_SPEC, $1, NIL, $2 ) ;
                }
              | opcode_spec args_spec arglist_spec
                {
                 /* Put in all elements */
                 $$ = mknode ( N_SPEC, $1, $2, $3 ) ;
                }
              ; /* End of rule for <spec> */
opcode_spec     : T_OPCODE opc_text ',' opc_size ',' nargs
                {
                 /* Put in null elements for name */
                 $$ = mknode ( N_OPCODE_SPEC, NIL, $2, $4, $6 ) ;
                }
              | T_OPCODE opc_name ':' opc_text ',' opc_size ',' nargs
                {
                 /* Put in all the elements for name */
                 $$ = mknode ( N_OPCODE_SPEC, $2, $4, $6, $8 ) ;
                }
              ; /* End of rule for <opcode_spec> */
opc_name        : T_NAME
                {
                 /* Make up a namenode */
                 $$ = mkname ( yytext ) ;
                }
```

```
            ; /* End of rule for <epc_name> */
epc_text      : expression
                /* An expression - it will do on its own */
            ; /* End of rule for <epc_text> */
epc_size      : expression
                /* An expression - it will do on its own */
            ; /* End of rule for <epc_size> */
nargs         : expression
                /* An expression - it will do on its own */
            ; /* End of rule for <nargs> */
args_spec     : T_ARGS arg_size
                {
                /* The name is NIL here */
                $$ = mknode ( N_ARGS_SPEC, NIL, $2 ) ;
                }
            | T_ARGS arg_name ':' arg_size
                {
                /* A full spec */
                $$ = mknode ( N_ARGS_SPEC, $2, $4 ) ;
                }
            ; /* End of rule for <args_spec> */
arg_name      : T_NAME
                {
                /* A name */
                $$ = mkname ( yytext ) ;
                }
            ; /* End of rule for <arg_name> */
arg_size      : expression
                /* An expression - it will do on its own */
            ; /* End of rule for <arg_size> */
arglist_spec  : T_ARGLIST arglist_count ',' arglist_el_size
                {
                /* A NIL name in this case */
                $$ = mknode ( N_ARGLIST_SPEC, NIL, $2, $4 ) ;
                }
            | T_ARGLIST arglist_name ':' arglist_count ','
                          arglist_el_size
                {
                /* A real name in this case */
                $$ = mknode ( N_ARGLIST_SPEC, $2, $4, $6 ) ;
                }
            ; /* End of rule for <arglist_spec> */
arglist_name  : T_NAME
                {
```

```
                /* Just a name */
                $$ = mkname ( yytext ) ;
               }
            ; /* End of rule for <arglist_name> */
arglist_count     : expression
                /* This is an expression - nothing to do */
            ; /* End of rule for <arglist_count> */
arglist_el_size   : expression
                /* This is an expression - nothing to do */
            ; /* End of rule for <arglist_el_size> */
rules           : rule
                {
                 /* A list of one */
                 $$ = mknode ( N_RULES, $1, NIL ) ;
                }
            | rule rules
                {
                 /* A list of rules */
                 $$ = mknode ( N_RULES, $1, $2 ) ;
                }
            ; /* End of rule for <rules> */
rule          : T_RULE rule_header rule_body T_ENDRULE
                {
                 /* A single rule */
                 $$ = mknode ( N_RULE, $2, $3 ) ;
                }
            ; /* End of rule for <rule> */
rule_header     : rule_name decl_list
                {
                 /* A full spec */
                 $$ = mknode ( N_RULE_HEADER, $1, $2 ) ;
                }
            ; /* End of rule for <rule_header> */
rule_name       : T_NAME
                {
                 /* Just a name */
                 $$ = mkname ( yytext ) ;
                }
            ; /* End of rule for <rule_name> */
decl_list       :
                {
                 /* No declaration, return NIL */
                 $$ = NIL ;
```

```
            }
          | decl decl_list
            {
             /* A list of declarations */
             $$ = mknode ( N_DECL_LIST, $1, $2 ) ;
            }
          ; /* End of rule for <decl_list> */
decl        : type variable ';'
            {
             /* A typed list */
             $$ = mknode ( N_DECL, $1, $2 ) ;
            }
          ; /* End of rule for <decl> */
type        : T_INT
            {
             /* Do nothing */
             $$ = T_INT ;
            }
          | T_STRING
            {
             /* Do nothing */
             $$ = T_STRING ;
            }
          | T_BOOL
            {
             /* Do nothing */
             $$ = T_BOOL ;
            }
          ; /* End of rule for <type> */
variable     : T_NAME
            {
             /* Just a name */
             $$ = mkname ( yytext ) ;
            }
          ; /* End of rule for <variable> */
rule_body    : match_part saving_part      generation_part
            {
             /* No spec in this rule body */
             $$ = mknode ( N_RULE_BODY, $1, $2, NIL, $3 ) ;
            }
          | match_part saving_part spec_part generation_part
            {
             /* Full rule body */
             $$ = mknode ( N_RULE_BODY, $1, $2, $3, $4 ) ;
            }
```

```
            ; /* End of rule for <rule_body> */
match_part      : T_MATCH '{' match_body '}'
                {
                 /* A match part is just its body */
                 $$ = $3 ;
                }
            ; /* End of rule for <match_part> */
match_body      : match_item
                {
                 /* Nothing to do here */
                 $$ = mknode ( N_MATCH_BODY, $1, NIL ) ;
                }
            | match_item ';' match_body
                {
                 /* Build up a list */
                 $$ = mknode ( N_MATCH_BODY, $1, $3 ) ;
                }
            ; /* End of rule for <match_body> */
match_item      : opcode_match
                {
                 /* No arg or arglist match */
                 $$ = mknode ( N_MATCH_ITEM, $1, NIL, NIL ) ;
                }
            | opcode_match arg_match
                {
                 /* No arg or arglist match */
                 $$ = mknode ( N_MATCH_ITEM, $1, $2, NIL ) ;
                }
            | opcode_match       arglist_match
                {
                 /* No arg or arglist match */
                 $$ = mknode ( N_MATCH_ITEM, $1, NIL, $2 ) ;
                }
            | opcode_match arg_match arglist_match
                {
                 /* No arg or arglist match */
                 $$ = mknode ( N_MATCH_ITEM, $1, $2, $3 ) ;
                }
            ; /* End of rule <match_item> */
opcode_match    : T_OPCODE opc_name
                {
                 /* Just the name */
                 $$ = $2 ;
                }
            ; /* End of rule for <opcode_match> */
```

```
arg_match       : T_ARG   arg_name
                {
                 /* An arg match is really just a name */
                 $$ = $2 ;
                }
              ; /* End of rule for <arg_match> */
arglist_match     : T_ARGLIST arglist_name
                {
                 /* An arglist match is really just a name */
                 $$ = $2 ;
                }
              ; /* End of rule for <arglist_match> */
saving_part       : T_SAVING '{' saving_body '}'
                {
                 /* Just a body */
                 $$ = $3 ;
                }
              ; /* End of rule for <saving_part> */
saving_body       : command_body
                {
                 /* Just pass back a command body */
                }
              ; /* End of rule for <saving_body> */
command_body      : decl_list command_list
                {
                 /* A command body */
                 $$ = mknode ( N_COMMAND_BODY, $1, $2 ) ;
                }
              ; /* End of rule */
command_list      : command
                {
                 /* A list of one. */
                 $$ = mknode ( N_COMMAND_LIST, $1, NIL ) ;
                }
              | command command_list
                {
                 /* A list of commands */
                 $$ = mknode ( N_COMMAND_LIST, $1, $2 ) ;
                }
              ; /* End of rule */
command         : assign_command
                {
                 /* Just return the command found */
                }
```

```
            | result_command
              {
               /* Just return the command found */
              }
            | if_command
              {
               /* Just return the command found */
              }
            | for_command
              {
               /* Just return the command found */
              }
            | do_command
              {
               /* Just return the command found */
              }
            | while_command
              {
               /* Just return the command found */
              }
            | '{' command_body '}'
              {
               /* Just return the command found */
               $$ = $2 ;
              }
            ; /* End of rule for <command> */
assign_command    : variable '=' expression ';'
              {
               /* Assignment command */
               $$ = mknode ( '=', $1, $3 ) ;
              }
            ; /* End of rule */
expression        : cond_expression
              {
               /* Return itself */
              }
            | arith_expression
              {
               /* Return itself */
              }
            ; /* End of rule */
cond_expression   : conditional T_COND expression ',' expression
              {
               /* An expression node */
               $$ = mknode ( T_COND, $1, $3, $5 ) ;
              }
            ; /* End of rule */
```

```
conditional     : '(' expression ')'
                {
                 /* Return the expression */
                 $$ = $2 ;
                }
              ; /* End of rule */
arith_expression   : expression T_LOGOR expression
                {
                 $$ = mknode ( T_LOGOR, $1, $3 ) ;
                }
              | expression T_LOGAND expression
                {
                 $$ = mknode ( T_LOGAND, $1, $3 ) ;
                }
              | expression T_EQ   expression
                {
                 $$ = mknode ( T_EQ, $1, $3 ) ;
                }
              | expression T_NE   expression
                {
                 $$ = mknode ( T_NE, $1, $3 ) ;
                }
              | expression '<'    expression
                {
                 $$ = mknode ( '<', $1, $3 ) ;
                }
              | expression T_LE   expression
                {
                 $$ = mknode ( T_LE, $1, $3 ) ;
                }
              | expression '>'    expression
                {
                 $$ = mknode ( '>', $1, $3 ) ;
                }
              | expression T_GE   expression
                {
                 $$ = mknode ( T_GE, $1, $3 ) ;
                }
              | expression '+'    expression
                {
                 $$ = mknode ( '+', $1, $3 ) ;
                }
              | expression '-'    expression
                {
                 $$ = mknode ( '-', $1, $3 ) ;
                }
              | expression '*'    expression
                {
                 $$ = mknode ( '*', $1, $3 ) ;
                }
              | expression '/'    expression
```

```
                {
                 $$ = mknode ( '/', $1, $3 ) ;
                }
              | expression '%'    expression
                {
                 $$ = mknode ( '%', $1, $3 ) ;
                }
              | expression T_LSHIFT expression
                {
                 $$ = mknode ( T_LSHIFT, $1, $3 ) ;
                }
              | expression T_RSHIFT expression
                {
                 $$ = mknode ( T_RSHIFT, $1, $3 ) ;
                }
              | '!' expression
                {
                 $$ = mknode ( '!', $2 ) ;
                }
              | '-' expression  %prec T_UMINUS
                {
                 $$ = mknode ( T_UMINUS, $2 ) ;
                }
              | primary
                {
                 /* Return itself */
                }
              ; /* End of rule for <arith_expression> */
primary         : variable
                {
                 /* Return itself */
                }
              | selection
                {
                 /* Return itself */
                }
              | function_ap
                {
                 /* Return itself */
                }
              | constant
                {
                 /* Return itself */
                }
              | '(' expression ')'
                {
                 /* Return the expression */
                $$ = $2 ;
                }
              ; /* End of rule */
```

```
selection       : variable '.' selector
                {
                 /* A selection */
                 $$ = mknode ( N_SELECTION, $1, NIL, $3 ) ;
                }
              | variable '[' expression ']' '.' selector
                {
                 /* A selection of an arg */
                 $$ = mknode ( N_SELECTION, $1, $3, $6 ) ;
                }
              ; /* End of rule for <selection> */
selector        : T_TEXT
                {
                 /* Return itself */
                 $$ = T_TEXT ;
                }
              | T_SIZE
                {
                 /* Return itself */
                 $$ = T_SIZE ;
                }
              | T_ID
                {
                 /* Return itself */
                 $$ = T_ID ;
                }
              | T_VALUE
                {
                 /* Return itself */
                 $$ = T_VALUE ;
                }
              | T_NARGS
                {
                 /* Return itself */
                 $$ = T_NARGS ;
                }
              ; /* End of rule for <selector> */
constant        : T_NUMBER
                {
                 /* Make a number node */
                 $$ = mknum ( yylval ) ;
                }
              | T_STRLITERAL
                {
                 /* Make a string node */
```

```
                $$ = mkstr ( yytext ) ;
                }
            | T_TRUE
                {
                /* A true node */
                $$ = mknode ( T_TRUE ) ;
                }
            | T_FALSE
                {
                /* A false node */
                $$ = mknode ( T_FALSE ) ;
                }
            ; /* End of rule for <constant> */
function_ap     : function_name '(' ')'
                {
                /* No args */
                $$ = mknode ( N_FUNCTION_AP, $1, NIL ) ;
                }
            | function_name '(' expr_list ')'
                {
                /* Arg list */
                $$ = mknode ( N_FUNCTION_AP, $1, $3 ) ;
                }
            ; /* End of rule for <function_ap> */
function_name   : T_NAME
                {
                /* Just make a name node */
                $$ = mkname ( yytext ) ;
                }
            ; /* End of rule for <function_name> */
expr_list       : expression
                {
                /* A list of one */
                $$ = mknode ( N_EXPR_LIST, $1, NIL ) ;
                }
            | expression ',' expr_list
                {
                /* A list of expressions */
                $$ = mknode ( N_EXPR_LIST, $1, $3 ) ;
                }
            ; /* End of rule for <expr_list> */
result_command  : T_RESULT expression ';'
                {
                /* A RESULT command */
                $$ = mknode ( T_RESULT, $2 ) ;
                }
```

```
            ; /* End of rule */
if_command      : T_IF conditional command
                {
                 /* An IF command with no ELSE */
                 $$ = mknode ( T_IF, $2, $3, NIL ) ;
                }
            | T_TEST conditional command T_ELSE command
                {
                 /* An IF command with an ELSE */
                 $$ = mknode ( T_IF, $2, $3, $5 ) ;
                }
            ; /* End of rule for <if_command> */
for_command     : T_FOR variable start_expr T_TO end_expr command
                {
                 /* A FOR node */
                 $$ = mknode ( T_FOR, $2, $3, $5, $6 ) ;
                }
            ; /* End of rule for <for_command> */
start_expr      : '(' expression ')'
                {
                 /* Just itself */
                 $$ = $2 ;
                }
            ; /* End of rule for <start_expr> */
end_expr        : '(' expression ')'
                {
                 /* Just itself */
                 $$ = $2 ;
                }
            ; /* End of rule for <end_expr> */
do_command      : T_DO command T_WHILE conditional
                {
                 /* A loop tested at the end */
                 $$ = mknode ( T_DO, $2, $4 ) ;
                }
            ; /* End of rule for <do_command> */
while_command   : T_WHILE conditional command
                {
                 /* A loop tested at the end */
                 $$ = mknode ( T_WHILE, $2, $3 ) ;
                }
            ; /* End of rule <while_command> */
```

```
generation_part    : T_GENERATION conditional '{' generation_seq '}'
                   {
                    /* Now the generation code */
                    $$ = mknode ( T_GENERATION, $2, $4 ) ;
                   }
                 ; /* End of rule for <generation_part> */
generation_seq     :
                   {
                    /* Nothing is NIL */
                    $$ = NIL ;
                   }
                 | generation_item generation_seq
                   {
                    /* List of generation items */
                    $$ = mknode ( N_GENERATION_SEQ, $1, $2 ) ;
                   }
                 ; /* End of rule for generation_seq> */
generation_item    : opcode_gener args_gener
                   {
                    /* No arglist */
                    $$ = mknode ( N_GENERATION_ITEM, $1, $2, NIL ) ;
                   }
                 | opcode_gener        arglist_gener
                   {
                    /* No args of arglist */
                    $$ = mknode ( N_GENERATION_ITEM, $1, NIL, $2 ) ;
                   }
                 | opcode_gener args_gener arglist_gener
                   {
                    /* No args of arglist */
                    $$ = mknode ( N_GENERATION_ITEM, $1, $2, $3 ) ;
                   }
                 ; /* End of rule for <generation_item> */
opcode_gener       : T_OPCODE opc_name
                   {
                    /* Really just a name */
                    $$ = $2 ;
                   }
                 ; /* End of rule */
args_gener         : T_ARGS          arg_value
                   {
                    /* NIL name */
                    $$ = mknode ( N_ARGS_GENER, NIL, $2 ) ;
                   }
                 | T_ARGS arg_name ':' arg_value
                   {
```

```
                /* NIL name */
                $$ = mknode ( N_ARGS_GENER, $2, $4 ) ;
               }
            ; /* End of rule for <args_gener> */

arg_value       : expression
               {
                /* Just itself */
               }
            ; /* End of rule for <arg_value> */

arglist_gener    : T_ARGLIST           arglist_value
               {
                /* NIL name */
                $$ = mknode ( N_ARGLIST_GENER, NIL, $2 ) ;
               }
            | T_ARGLIST arglist_name ':' arglist_value
               {
                /* name */
                $$ = mknode ( N_ARGLIST_GENER, $2, $4 ) ;
               }
            ; /* End of rule */

arglist_value    : expression
               {
                /* Just itself */
               }
            ; /* End of rule */

%%

/* Useful routines */



int mknode ( type, a1, a2, a3, a4, a5, a6 )

int type ;                    /* Node id */
int a1, a2, a3, a4, a5, a6 ;

/* Make a new treenode. This is all rather horrible, since the compiler
  expects integers to be passed around. We solve the problem by casting
  to and from integers. */

 {
  struct treenode *node = get_treenode () ;

  node -> type = type ;
  node -> a1   = ( struct treenode * ) a1 ;
  node -> a2   = ( struct treenode * ) a2 ;
  node -> a3   = ( struct treenode * ) a3 ;
  node -> a4   = ( struct treenode * ) a4 ;
  node -> a5   = ( struct treenode * ) a5 ;
  node -> a6   = ( struct treenode * ) a6 ;

  return ( int ) node ;           /* This is APPALLING */

 } /* mknode ( type, a1, a2, a3, a4, a5, a6 ) */
```

```
int mkname ( text )
char *text ;
/* Make the name permanent, and return a name node */
 {
  char *ptext = get_str ( text ) ;
  return mknode ( T_NAME, ( int ) ptext ) ;
 } /* mkname ( text ) */


int mknum ( value )
int value ;
/* Set up a value node */
 {
  return mknode ( T_NUMBER, value ) ;
 } /* mknum ( value ) */


int mkstr ( text )
char *text ;
/* Make the string permanent, and return a literal node */
 {
  char *ptext = get_str ( text ) ;
  return mknode ( T_STRLITERAL, ( int ) ptext ) ;
 } /* mkstr ( text ) */
```

Appendix C. Proof of Minimisation

This is the proof of the minimisation of code size used in section 5.3 and derived by Smith [Bennett87].

Smith uses the method of Lagrange Multipliers to maximise the saving. Let s_i, x_i and t be defined as in chapter five. Let us suppose there exist non-negative constants a_i, such that:

$s_i = \sigma f_i (1-e^{-a_i x_i})$, where σ is a suitably dimensioned constant.

Thus the total space saved is:

$$S = \sigma \sum_{i=1}^{t} f_i (1 - e^{-a_i x_i}) \qquad \text{(Eqn. 1)}$$

subject to the constraint $\sum x_i = c$, where c is the number of special instructions we have room for in our instruction set of 256 instructions.

Let $g = \sum_{i=1}^{t} x_i - c$ and S be given by Eqn 1.

Let $\psi = S + \lambda g$

We must solve simultaneously

$$\nabla\psi = \underline{0} \quad \text{and} \quad g = 0$$

Thus

$$a_i f_i e^{-a_i x_i} = a_j f_j e^{-a_j x_j} \quad \text{for all } i, j \in \{1, \ldots, t\}$$

Let $d_i = \prod_{\substack{j=1 \\ j \neq i}}^{t} a_j$, then for any $k \in \{1, \ldots, t\}$ we have

$$\prod_{i=1}^{t} \left[a_k f_k e^{-a_k x_k}\right]^{d_i} = \prod_{i=1}^{t} \left[a_i f_i e^{-a_i x_i}\right]^{d_i} .$$

Therefore

$$\left[a_k f_k e^{-a_k x_k}\right]^{\sum_{i=1}^{t} d_i} = \prod_{i=1}^{t} \left[a_i f_i\right]^{d_i} . e^{-a \sum_{i=1}^{t} x_i}$$

where

$$a = \prod_{i=1}^{t} a_i .$$

Now $g = 0$ so $\sum_{i=1}^{t} x_i = c$

Let $b = \sum_{i=1}^{t} d_i$.

Now we have

$$\left[a_k f_k e^{-a_k x_k}\right]^{b} = \prod_{i=1}^{t} \left[a_i f_i\right]^{d_i} . e^{-ac}$$

Let r denote the right hand side of this equation (which is independent of k). Let $\eta_k = (a_k f_k)^b$ and then

$$\eta_k e^{-b a_k x_k} = r$$

and finally we have

$$x_k = \frac{1}{b a_k} \log_e \left(\frac{\eta_k}{r}\right) \qquad \text{(Eqn. 2)}$$

Such a choice of x_i 's will maximise S.

The method of Lagrange multipliers ensures that we have found a critical point of the saving function. This is in fact a maximum, as may be seen by considering the problem in question, although Smith does not give a proof of this.