Technical Report

UCAM-CL-TR-93
ISSN 1476-2986

Number 93

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Automated design of an instruction set for BCPL

J.P. Bennett

July 1986

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

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

© 1986 J.P. Bennett

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

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-93*

Automated Design of an Instruction Set for BCPL

J.P. Bennett

A number of attempts have been made over the years to use observations of the use of machine instruction sets, as a design tool for improved instruction sets. This report presents a methodology for using statistical analysis of compiled code as a basis for the design of an instruction set suited to a particular environment. The design of an instruction set for BCPL within the TRIPOS operating system is presented as a case study.

Contents

1.	Introduction	1
1.1.	Previous Work	1
1.2.	Methodology	6
2.	The BCPL Instruction Set	9
2.1.	A Canonical Instruction Set	9
2.2.	Analysis of the Canonical Instruction Set	12
2.3.	The Design Rules	13
2.4.	Instruction Set Generation	15
2.5.	Testing the New Instruction Set	18
3.	Future Work	23
3.1.	Limitations of the Current System	23
3.2.	Application to Other Language Systems	24
3.3.	A More General System	24
4.	Bibliography	27
Appendix I	Instruction Set Generation	29
Appendix II	Peephole Optimisation	43

1. Introduction

New techniques, together with more widespread application of microprogramming are giving ever greater flexibity in the design of computer architectures. With this flexibility comes a need to formalise the reasoning behind decisions taken during design. This report suggests a way of choosing the instruction set for a new computer.

1.1. Previous Work

Almost all instruction sets have been designed to assist the compiler writer. In most cases these efforts have been based on intuitive assessment by the designer of what is needed. Thus to help compilers many machines have opcodes to assist handling of high-level language stacks or complex floating point operations (e.g. the VAX-11 POLYD opcode for handling polynomials [DEC81]). However such design efforts have been limited by the desire to provide a simple, orthogonal machine language for assembler programmers.

Such designs have been of use to assembly programmers, but have not always been successful as targets for compiled high-level languages. The problem is that ever more complex opcodes become harder to map exactly onto high-level constructs. A typical example is that Cambridge LISP for the VAX-11 compiles its procedure calls using a number of simple operations, rather than one of the VAX-11 call opcodes, which do not map correctly onto LISP's model of procedure calling, and is hence slower. In addition the orthogonal nature of most instruction sets, whilst providing flexibility for the assembler programmer, means most of the instruction set is not used by the typical compiler.

Over the years many examinations of precisely what code compilers do produce from typical high level programs have been carried out. Knuth [Knuth71] examined FORTRAN, and discovered that most

expressions had one operator, and one operand, and most loops were not nested. An excellent survey of work in this area can be found in [Weicker84].

A number of designers have attempted to look at the code compiled for high-level languages, and use this to assist in the design of instruction sets more suited to the needs of compilation for a particular high level language. A good early attempt was the experimental machine, EM-1, designed by Tanenbaum [Tanenbaum78]. He analysed both the static size (i.e. the space occupied by the compiled code) and dynamic frequency (i.e. how often each opcode was executed) of compiled SAL (a block-structured, typeless and GOTO-less language), and used the ideas gleaned from this to propose an instruction set for the language. This had addressing modes to facilitate use of local and global variables, which he observed were the most commonly used, and simple stack arithmetic. The instruction set had 16-bit addressing and was based on a byte code architecture, i.e. opcodes and arguments as multiples of whole bytes. Particularly common opcode-argument pairs were made into single opcodes (e.g. PUSH LOCAL 0), common pairs of opcodes were made into single opcodes (e.g. PUSH 0 ; POP X became ZERO X), opcodes which commonly had small (<= 255) argument values had new versions with byte, rather than word arguments, and rare opcodes were relegated to 2-byte opcodes.

Tanenbaum used his information in an informal manner to guide his choice of opcodes, and was aiming at an improvement both in static code size and dynamic opcode frequency. He gave figures suggesting a 2 or 3-fold saving in static code size over PDP-11 and Cyber machines.

A system with a more precise objective was the CINTCODE system for BCPL [Richards82]. This was designed to be used as an interpretive byte code on exisiting mini and micro-computers, and its design criterion was to minimise static size of compiled code. It did not make use of any statistics of the sort used by Tanenbaum, but used the knowledge of an experienced BCPL compiler writer, who was aware of the operations he most needed. Like EM-1 this instruction set was based on a machine with byte-sized opcodes and 16-bit addressing. To facilitate comparison with modern mainframe architectures a version of CINTCODE with 32-bit addressing was deriveded. A compiler for this was used to collect statistics on 100 programs (over 1000 BCPL procedures), used as commands in the TRIPOS operating system. Table 1.1. shows the distribution of opcodes found in this code when compiled, by comparison with the distribution of opcodes provided in the instruction set (247 1-byte opcodes), grouped by type. We should expect these two sets of figures to be similar in a well designed instruction set. Although there are obvious anomilies, the similarities are striking, particularly given

Instruction Type	*Instr. Set Freq.*		*Compiled Freq.*	
Load Local	17.8%	32.0%	17.0%	39.5%
Load Global	6.1%		6.9%	
Load Static	1.6%		4.3%	
Load Immediate	6.5%		11.3%	
Store Local	9.3%	14.6%	21.8%	26.2%
Store Global	2.4%		2.2%	
Store Static	0.8%		0.1%	
Store Register	2.0%		2.1%	
Jump Unconditional	1.2%	10.9%	4.9%	11.1%
Jump Conditional	9.7%		6.2%	
Call Register	4.9%	17.0%	4.7%	7.6%
Call Global	12.1%		2.8%	
Operate on Local	10.9%	14.6%	4.0%	11.9%
Operate on Global	4.9%		2.4%	
Operate on Immediate	1.2%		4.3%	
Operate on Register	4.5%		1.2%	
Miscellaneous	4.0%	4.0%	3.7%	3.7%

Table 1.1. 32-bit CINTCODE Opcodes

that the compiler that generates this code was not written by the person who designed the original CINTCODE. That it is effective can be seen from Table 1.2., which compares CINTCODE with code compiled for two other machines and OCODE (the intermediate code for BCPL and the original inspiration for CINTCODE). Although figures such as these are helpful it should be born in mind that the most wonderful instruction set in the world will not make up for a poor compiler design, and far more detailed analysis is needed before precise statements about relative code statements can be made. As an example Table 1.3. shows the relative sizes of OCODE and MC68000 code for three popular local compilers. It should be noted that the figure for TBCPL OCODE, a commercial compiler for which we have no source is suspect.

Target Machine	*Relative Code Size*
IBM 370	1.00
MC68000	0.84
OCODE	0.65
CINTCODE	0.52

Table 1.2. Static Code Size for Compiled BCPL

Compiler	*OCODE*	*MC68000*
BCPL	0.76	0.98
BCP	0.38	1.00
TBCPL	0.12?	0.96

Table 1.3. Variation in Compiler Performance

In each of these cases knowledge of the sort of operations required by a compiler writer for a particular language was used to influence the design of an instruction set. In the case of EM-1 the instruction set was designed from scratch, in the case of CINTCODE it was a refinement of an existing instruction set, OCODE. In neither case was the statistical information obtained used to provide either precise prediction of the improvements to be gained, or to drive the selection of new opcodes.

Sweet and Sandman went one stage further in their work refining the instruction set of the Alto [Sweet82]. The Alto has an instruction set designed to support compilation of the MESA language. It was originally designed, in a similar way to CINTCODE, using the experience of a number of compiler writers to suggest relevant opcodes. It too is a byte-coded machine with 16-bit addressing. It became apparent that the machine was running out of address space for programs, and an attempt was made to refine the instruction set to reduce static code size. In their work Sweet and Sandman introduce the idea of *normalisation*. They took all opcodes which were combinations of existing opcodes, or which had combined arguments and broke them down into their constitutent opcodes, with all arguments explicitly stated. For example the opcode to skip on eight bytes, `Jump8` became `Jump` with a 2 byte opcode, value 8 and `JumpNotZero` became `Load 0, JumpNot Equal`. This gave them a simple instruction set with around 100 opcodes, and about 2.5 million

opcodes to analyse. They collected the following five sorts of information:

A frequency histogram of all opcodes

Histograms of argument distribution for each opcodes arguments

Frequency distributions of the succeeding opcode for each opcode.

Frequency distributions of the preceding opcode for each opcode.

Frequency distributions of all opcode pairs.

Note that although the last three give the same information in total, they show the information in the different ways that may be useful (depending on whether you are interested in conditional probability of one opcode on another, or the frequency of a particular opcode pair). Using this information they suggested a new set of instructions, derived by combining opcodes and arguments to give opcodes with implied arguments and combining opcode pairs. They then took the numerical information in their data, and were able to predict a reduction of 12% in static code size for this new instruction set. A peephole optimiser was used to generate this new code, and revealed precisely this saving. Although they used their numerical information to predict the results of their proposals, there is no indication of the choice of new opcodes being driven directly from this data.

These methods have in general relied on trying to set up instruction sets with operations allied to those required by the high-level language. An alternative approach has been followed by the designers of RISC architectures [e.g. Patterson81, Acorn85]. These machines have extremely simple fast instruction sets, the aim being that a combination of several of these can match precisely any high-level operation. In a sense the burden of matching high-level construct to low-level operation is now moved from the microprogrammer, to the compiler writer. It should be noted that this is an attempt to solve the same problem as that faced by the designers of the instruction sets previously described (so-called CISC architectures), but the solution is completely different. There are arguments both for and against this approach, but these will not be discussed here. The aim of this report is to demonstrate a new technique for designing CISC machines.

1.2. A New Methodology

The new methodology for designing computer instruction sets is based on statistical analysis of compiled code. It is particularly flexible, capable of defining completely new instruction sets, or refining existing instruction sets to meet various criteria. Typical criteria would be the minimisation of static code size or dynamic opcode frequency, but there is no reason why other criteria, such as manufacturing cost, or even combinations of criteria could not be used.

There are three stages in designing a new instruction set this way:

1. Design a "Canonical Instruction Set"

2. Collect statistics from a compiler for this canonical instruction set.

3. Apply rules to generate new opcodes from the canonical instruction set, using the statistics gained from stage 2.

It is anticipated that one high-level language is being catered for, and that opcodes are to be provided that can be microcoded on a particular target architecture. In the case discussed later the high-level language involved is BCPL, and the target microarchitecture a High Level Hardware ORION minicomputer. Extensions to this basic philosophy, particularly to handle multiple language systems, are considered later.

The canonical instruction set is designed by providing one opcode, or set of opcodes to map onto each construct in the high-level language. For most constructs this will involve only one opcode, but some, such as loops may require more than one (e.g. an opcode at each end). These opcodes must be in terms that can be supported by the target architecture, so direct translation of a particular high level construct is not always possible. After this has been carried out it may turn out that some canonical opcodes are identical, for instance an IF opcode, may have an identical effect to the opcode that starts a WHILE loop (the WHILE loop needing a second opcode at its other end). Such duplicates are eliminated.

A compiler is written for the canonical instruction set, and used to generate code. It would be wise at this stage to provide an interpreter for the canonical instruction set, so its validity can be checked. Code is compiled for all the programs to be used in the final system, or at least for a representative sample, and statistically analysed. Clearly

an interpreter will be needed to collect dynamic statistics. What statistics are collected depend very much on stage 3, and the criteria to which the instruction set is being designed.

The final stage is to specify design rules for new opcodes. These rules must be ways of taking existing opcodes and combining or modifying them to generate new opcodes that may improve the instruction set, and which could be implemented on the target architecture. Typical design rules are:

Combine two opcodes

Generate a new opcode for small argument values

Generate an opcode that has a specific argument value implied

Remove a little used opcode

These rules are then used to generate new opcodes using the statistics gathered already to guide the selection. This may be done in an arbitrarily complex manner, but a simple approach is to apply all rules to all possible targets, and select the new opcode which gives the best improvement in the target criteria (code size, dynamic frequency etc.), repeating this operation until enough opcodes are obtained. Statistics for new instructions must be derived from the existing statistics. Clearly this simple approach is likely to fall down under some circumstances, particularly if we provide rules that do not necessarily lead to an improvement on their own (For example even when the design criterion is minimising static code size we may provide a rule to eliminate redundant opcodes, which applied alone would not give a code improvement). More complex approaches have to be developed to get better results.

This system is flexible. To refine an existing instruction set we may either use the existing instruction set as the canonical instruction set, or a normalised version of it. (see the description of Sweet and Sandeman's work above). The system can cope with machines to support multiple languages, by starting with a canonical instruction set that maps on to the basic constructs of all the languages involved (many of which will overlap), and collecting compiled code statistics for all these languages. We can investigate effects of different design approaches by changing the canonical instruction set or the design rules.

This is not a fully automatic system, but an aid to design. The quality of the final product depends on the designer's skill in choosing his canonical instruction set and in specifying appropriate design rules.

However, within these limitations it should provide a method of efficiently designing machines that approach the best possible.

2. An Instruction Set for BCPL

The new methodology has been applied to derive an instruction set suitable for use with BCPL in a TRIPOS environment. TRIPOS is a small real-time operating system written in BCPL and is described in [Richards79]. It should be noted that it is important to constrain the environment being supported to the system as well as the language from which statistical data are taken, since even within one language there may be considerable variation in use in different settings (see [Knuth71] for an example). The design criterion is the minimisation of static code size, and the target architecture is a High Level Hardware ORION minicomputer, a 32-bit word addressed microprogrammable machine of about the power of a VAX-11/750 [HLH85].

2.1. A Canonical Instruction Set

There appear to be three elements in a simple imperative language such as BCPL:

Access to data of various sorts (local, global etc.)

Manipulation of such data (addition indirection etc.)

Flow of control

To some extent the first two overlap, for example access to the address of a variable. Elements in the instruction set are required to model each of these concepts.

There are four (or possibly only three) types of data within BCPL:

Global — This is a contiguous area. A suitable model of access for the ORION is as a base register and constant offset.

Local This too is a contiguous area within the extent of a procedure. A suitable model is by base register and constant offset. However in this case the base register value changes across procedure calls.

Static This can be modeled in a similar manner to global data. However we must ensure that linking information is provided to enable several sections to share the data area.

Immediate This is normally viewed as just an argument field for an opcode, but could possibly be considered as a type of immutable static data (something I shall not do).

It is necessary to provide methods of loading from and storing to these locations in word sized chunks, together with a facility for loading addresses of locations in these areas. BCPL's indirection operators also require us to be able to load from and store to computed addresses in both word and byte sized chunks.

BCPL's model of expressions is as black boxes. Arbitrary combinations of data items yield a new data item. It is not feasible to provide an opcode for each possible expression, and so a simple model of evaluation must be used. Previous instruction sets have used registers or the top of the BCPL stack. The model chosen is a finite sized scratchpad stack, together with a stack operator for each operator in BCPL. This is adequate apart from the problem of expressions involving recursive functions (which may be embedded within VALOF blocks), which allow the building of expression stacks of arbitrary depth, which depth cannot be computed at compile time. It must be accepted that the contents of the scratchpad stack are lost across function calls and VALOF blocks, and anything on the scratchpad stack will need to be saved on the BCPL stack on such occasions. Thus, the model is of data access opcodes which push data to and pop data from a scratchpad stack, and arithmetic operators which manipulate data on the top of this stack. This is not perfect because of the recursion problem, and combined data access and manipulation opcodes must be provided to handle the loading of addresses of variables. However it is simple, and will serve as a basis for a first design.

Looking at the structure of BCPL it might be thought that flow of control is tree structured, and could be modelled by operations that push and pop start and end addresses of loops and procedures to and from a control stack. However the operations of RESUTLIS, procedure return and GOTO represent transfers of control that do not follow this tree-structured model, and so address information must be

put in the opcodes, rather than pushing and popping such information implictly on a "control stack". An opcode or opcode pair is provided for each of the branching and looping constructs in BCPL.

The canonical instruction set derived from this model is:

Data Access:

```
GLOBALPUSH, LOCALPUSH, STATICPUSH, IMMEDIATE,
PUSH, PUSHBYTE,
GLOBALPUSHLEFT, LOCALPUSHLEFT, STATICPUSHLEFT,
GLOBALPOP, LOCALPOP, STATICPOP,
POP, POPBYTE
```

Data Manipulation:

```
NEG, NOT, ABS,
MULT, DIV, REM, PLUS, MINUS, EQ, NE, LS, GR, LE,
GE, LSHIFT, RSHIFT, LOGAND, LOGOR, EQV, NEQV
```

Flow of Control:

```
CALL,
FOR, ENDFOR, REPEATUNTIL, REPEATWHILE, REPEAT,
WHILE, UNTIL,
BREAK, LOOP, RESULTIS,
RETURN, GOTO, FINISH,
IF, UNLESS, TEST, ELSE,
SWITCH
```

These opcodes may have a number of arguments, typically one for data access, none for data manipulation, and up to three for flow of control opcodes. The full description of the instruction set is fairly long and available from the author if required.

A little examination reveals that the semantics of IF and WHILE are the same (the WHILE is balanced by a REPEAT later on), and similarly UNLESS and UNTIL are the same. BREAK, LOOP and RESULTIS are either the same as REPEAT, or a forward jump, for which we introduce the opcode JUMP. Finally IF is the same as TEST, and ELSE is the same as JUMP. Thus the following opcodes for flow of control remain:

```
CALL,
FOR, ENDFOR, REPEATUNTIL, REPEATWHILE, REPEAT,
IFWHILE, UNLESSUNTIL,
JUMP,
RETURN, GOTO, FINISH,
SWITCH
```

As a matter of choice I have decided to have all arguments unsigned, although hindsight suggests that in this case I should have had **IMMEDIATEMINUS** as an additional opcode. The initial representation chosen is one byte per opcode, and one word (4 bytes) per argument.

2.2. Analysis of the Canonical Instruction Set

A compiler for the new instruction set has been constructed. From it were tabulated the static frequency of instuctions, the static frequency distribution of arguments to opcodes, and a table of static frequencies of pairs of opcodes. The table of pair frequencies excluded those pairs that were split by a label (they would never be able to be joined into a single opcode). Table 2.1. shows the frequency of the

Instruction	*Frequency*	*Instruction*	*Frequency*
IMMEDIATE	20.2269%	LOGOR	0.1482%
LOCALPUSH	17.7474%	PUSHBYTE	0.1256%
GLOBALPUSH	15.1757%	STATICPOP	0.1052%
CALL	9.9152%	LE	0.1015%
LOCALPOP	9.7850%	GLOBALPUSHLEFT	0.0781%
IFWHILE	4.4183%	GR	0.0741%
JUMP	3.7594%	GOTO	0.0709%
GLOBALPOP	3.2136%	REPEATUNTIL	0.0621%
STATICPUSH	2.4795%	POPBYTE	0.0565%
STATICPUSHLEFT	2.3774%	LS	0.0509%
FOR	1.7211%	REPEATWHILE	0.0391%
ENDFOR	1.7211%	MULT	0.0361%
PLUS	1.2089%	NOT	0.0359%
UNLESSUNTIL	1.0255%	GE	0.0324%
EQ	0.7399%	DIV	0.0310%
REPEAT	0.6700%	RSHIFT	0.0300%
PUSH	0.6601%	LSHIFT	0.0202%
LOCALPUSHLEFT	0.4166%	QUERY	0.0198%
SWITCH	0.4146%	REM	0.0144%
RETURN	0.3367%	NEG	0.0138%
POP	0.2700%	ABS	0.0058%
LOGAND	0.2269%	FINISH	0.0024%
MINUS	0.1767%	NEQV	0.0014%
NE	0.1578%	EQV	0.0004%

Table 2.1. Canonical Instruction Set Frequency

opcodes, the rest of the data is verbose in the extreme in printed form, and is not included, but can be obtained from the author. Note that throughout this chapter reference to frequency of particular opcodes

means the frequency of the opcode weighted according to the size of the opcode *with its arguments*. The sample data were 102 BCPL programs that are commonly used as commands for the TRIPOS operating system. These generated 520 053 bytes of code, of which 20 768 bytes were due to jump tables for SWITCHON commands. These jump tables were excluded from the initial analysis, as a simplification, giving 499 285 bytes of code to be analysed. The tables being data, rather than opcodes are more complicated to handle than simple opcodes, and in addition their size is not fixed, being given as the first argument to the **SWITCH** opcode.

The BCPL compiler used is unconventional in that its code was generated direct from the AE tree, rather than from OCODE. I feel that the AE tree expresses the meaning of the original high-level program exactly, whereas OCODE has already imposed its own low-level view on the world. The code generator has naive optimisations, but little else (it is an experimental version), and so the code generated is not as optimal as it could be.

Quick observation of the results in Table 2.1. shows that there is general agreement with the trends found for CINTCODE.

2.3. The Design Rules

For this first attempt at demonstrating the method of instruction set design, just three design rules were used:

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

Create a new opcode with a single value of one argument implied. Thus **IMMEDIATEARG1=0** could be generated from **IMMEDIATE** to push constant zero onto the scratch stack.

Create a new opcode by concatenating two existing opcodes. For example **GLOBALPUSH** and **CALL** could be conbined to give **GLOBALPUSHCALL**, to call a global routine.

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

When an argument is shrunk, the suffix **BYTEARG*x***, **HALFWORDARG*x*** or **THREEBYTEARG*x*** is added, where *x* is the argument involved.

When an argument is combined with an opcode, the suffix **ARG**x=y is added, where x is the argument involved, and y, the value being combined.

When two opcodes are combined, the new name is the concatenation, in order of their two names.

Such names are inherently verbose, and may be ambiguous (consider for example **GLOBALPUSHBYTEARG1IMMEDIATEBYTEARG1ARG1=0**, where the 0 belongs to the **IMMEDIATE** part of the opcode. Was it **[GLOBALPUSHBYTEARG1] [IMMEDIATEBYTEARG1]ARG1=0**, or was it **[GLOBALPUSHBYTEARG1] [IMMEDIATEBYTEARG1ARG1=0]**), but examination of the entire generation listing can resolve these ambiguities. It is anticipated that rational names for these opcodes would be provided, once the set is decided upon.

It is necessary to consider the derivation of statistical information for new opcodes generated by each design rule.

In the case of shrinking argument size the new opcode will take all the cases of the old opcode for which the argument concerned will fit into the new argument size. This is adequate for opcodes with just one argument, but with multiple arguments there is the question of the distribution of the arguments not concerned in the shrinking. As a first approximation it is assumed arguments are independant, which may be so in many cases. Thus if a new opcode is generated, whose first argument is reduced to say one byte, thereby encompassing 75% of the occurrences of the existing opcode, it is assumed that all other arguments have their frequency distribution reduced to 25% of the original, for the original opcode, and take 75% of the original for the new opcode. A typical case where this is an oversimplification is in the **SWITCH** opcode, where a small value for the first argument (the number of cases), will tend to imply a small value for the second argument (the offset of the default label).

The adjustment of pair frequency data is similarly based on independency of argument value and the occurence of particular preceding or succeeding opcodes. If we consider the hypothetical case above again, where a new opcode, taking 75% of all opcodes is generated, it is assumed that the frequency of pairs involving this new opcode is 75% of the frequency of the equivalent pair involving the original opcode. Similarly pairs involving the original opcode are reduced to 25% of their previous value. This too is not always a valid model of the structure of compiled code.

Combining a particular argument value with an opcode is very similar to the first case. However, the method of calculating

frequencies of pair data, turns out to be very weak is some cases. Although pushing of global values may be followed by any number of operations, the operation of pushing global 73, which happens to be the global value of `writes` in this system is invariably followed by a `call` opcode. Similarly whilst subtraction of constants is common, and zero is a common constant, subtraction of zero never occurs, and thus if we create an opcode to push an immediate value of zero we will get an erroneous figure for the frequency of the pair involving **MINUS** and this new opcode.

When combining opcode pairs independence of argument values and opcode pairs is again assumed when calculating the new argument distributions.

2.4. Instruction Set Generation

My instruction set generator takes the very simple approach of trying each of the design rules on each existing opcode, and selecting the application which would give the greatest immediate reduction in code size. Each opcode is represented as a node giving details of the opcode (Fig. 2.1.), with a list of argument nodes, one for each

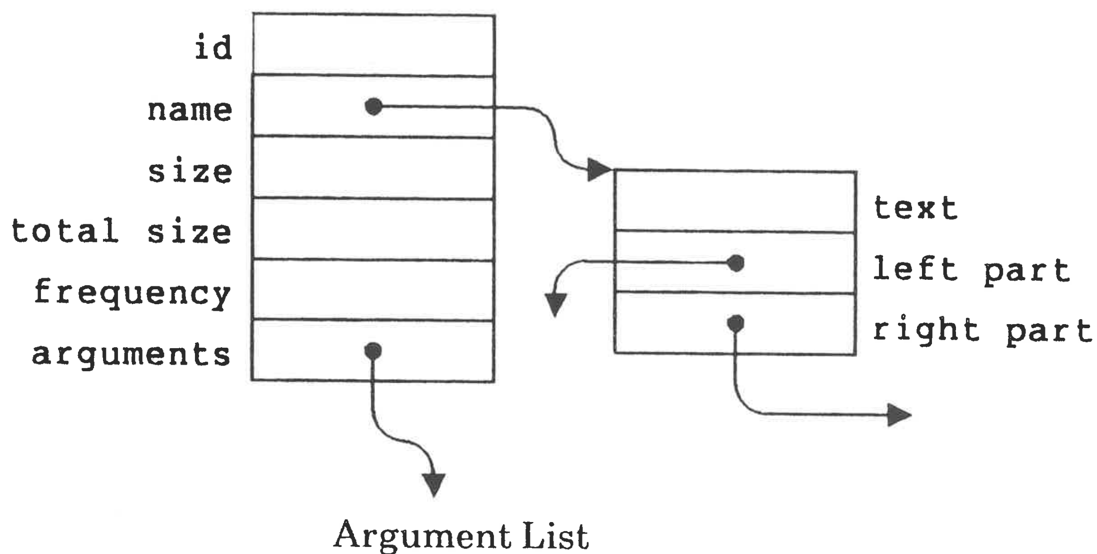


Fig. 2.1. Opcode Node with Name Node

argument. The tree structure provided for the name entry in an opcode node facilitates the building of names for new opcodes. The argument nodes contain details of the argument distribution as a histogram, with variable width bars to ensure at least a minimum fraction of the distribution is held in one bar (Fig. 2.2.). Opcode pair frequencies are held in a 2 dimensional array. For speed of

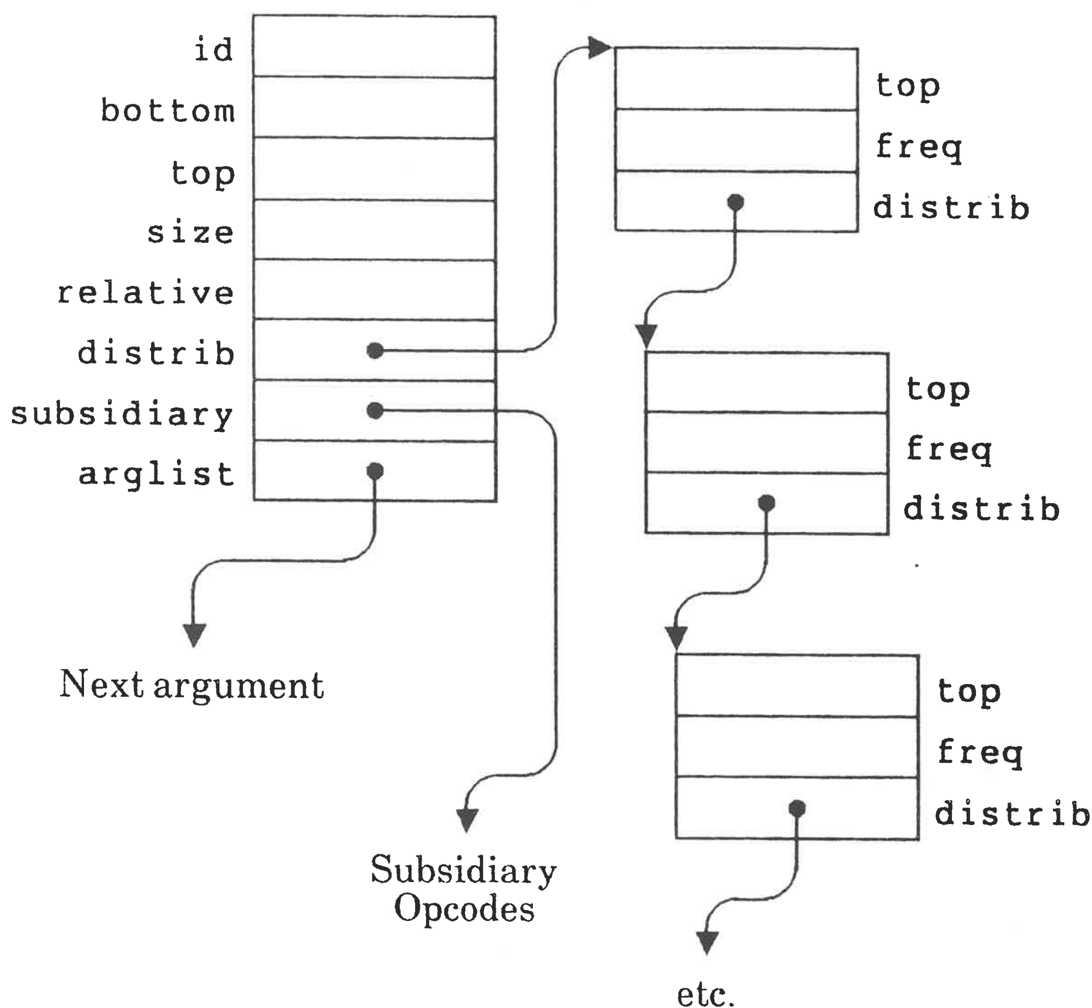


Fig. 2.2. Argument Node

calculation frequencies are held as integer numbers of opcodes, rather than floating point fractions. This makes for faster arithmetic, and also ensures care in handling rounding errors (the loss of a whole opcode is very noticeable, rather than errors of the order of 10^{-15} of an opcode as is the case with typical floating point calculations).

For each design rule two routines are provided, one to calculate the best reduction available by applying that rule, the other to actually apply the rule to a particular opcode and generate new statistical data. Each routine to try a particular rule hands back a data structure with details of the opcodes and/or arguments to be transformed to obtain the specified improvement in code size. The best saving is chosen, and the corresponding routine applied to generate the new data.

The opcode generation listing, giving each new opcode as it is generated, together with the saving in code size is shown in Appendix I . This also shows the resulting instruction set ordered by frequency, the first 24 results of which are summarised in Table 2.2. (*c.f.* Table 2.1.).

Instruction	*Freq.*
GLOBALPUSHBYTEARG1CALLBYTEARG1	4.03%
GLOBALPUSHBYTEARG1	2.89%
LOCALPUSHBYTEARG1	2.68%
IMMEDIATEBYTEARG1	2.43%
JUMPBYTEARG1	2.41%
LOCALPOPBYTEARG1	2.05%
GLOBALPUSHHALFWORDARG1CALLBYTEARG1	1.95%
GLOBALPOPBYTEARG1	1.88%
IMMEDIATEHALFWORDARG1	1.72%
IMMEDIATEBYTEARG1ARG1=0	1.70%
STATICPUSHLEFTHALFWORDARG1	1.69%
LOCALPUSHBYTEARG1ARG1=0	1.62%
STATICPUSHBYTEARG1CALLBYTEARG1	1.55%
JUMPHALFWORDARG1	1.47%
GLOBALPUSHHALFWORDARG1	1.47%
STATICPUSHLEFTHALFWORDARG1BYTEARG1	1.41%
GLOBALPUSHBYTEARG1IMMEDIATEBYTEARG1	1.38%
IMMEDIATEBYTEARG1ARG1=1	1.27%
LOCALPOPBYTEARG1LOCALPUSHBYTEARG1	1.27%
GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=0	1.23%
LOCALPUSHBYTEARG1ARG1=1	1.22%
GLOBALPUSHBYTEARG1IMMEDIATEBYTEARG1ARG1=0	1.06%
ENDFORARG2=1BYTEARG1BYTEARG3	0.98%
LOCALPUSHBYTEARG1ARG1=2	0.97%

Table 2.2. The Most Common New Opcodes

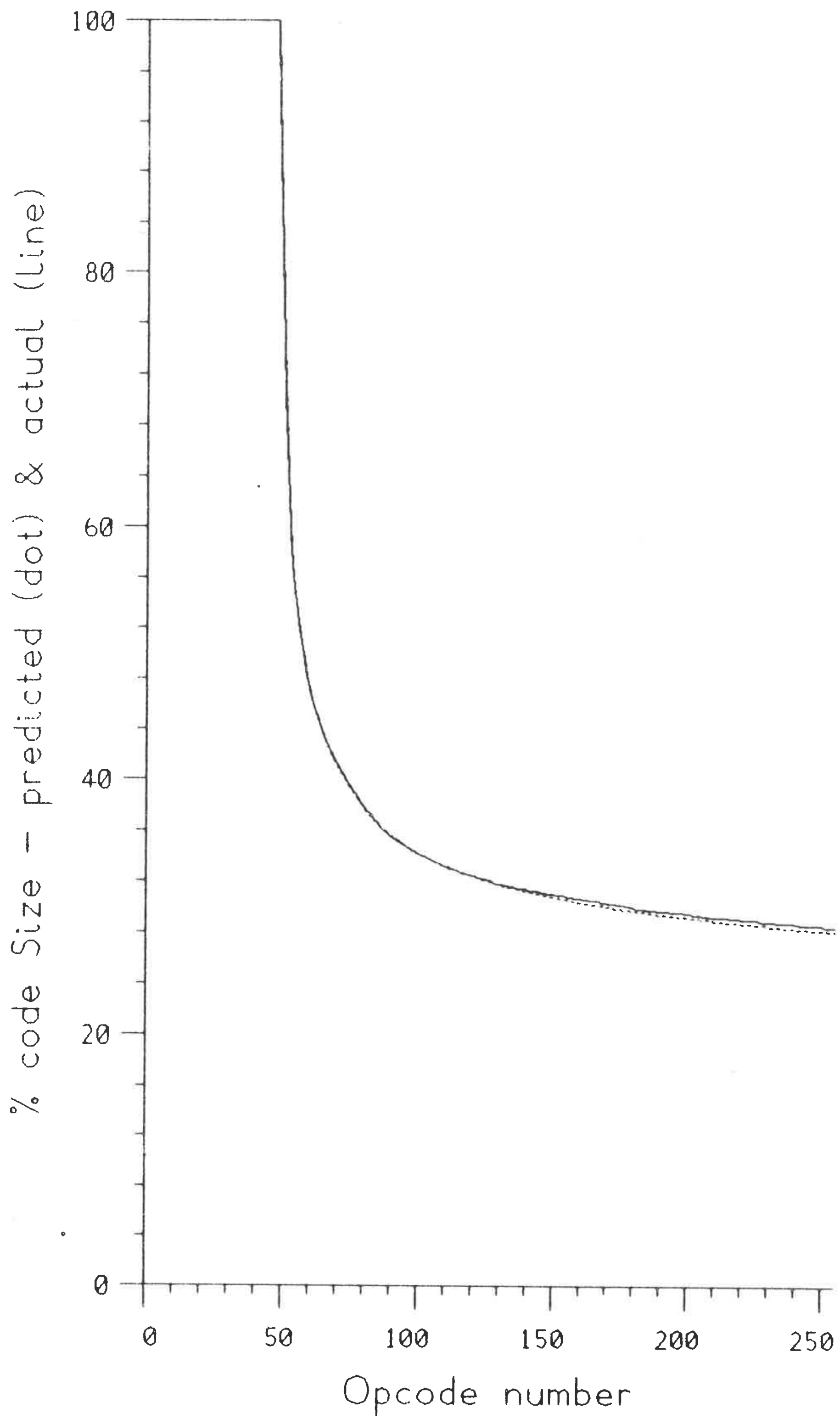
It can be seen that the new instruction set is far from orthogonal, as might be expected, and contains some surprising choices for opcodes. Some of these reflect the direct use of statistics in driving the system. For example we see that opcode 244 is **GLOBALPUSHBYTEARG1ARG1=73**. This is due to global 73 being the routine `writes` in the TRIPOS BCPL system, and this is part of the calling sequence for this very common routine. This underlines the comment made earlier that the instruction set suggested is very closely tied to the system from which the statistics were taken. In another system global 73 might very well not be `writes` and this opcode would be useless. Similarly it would be useless in any system that produces little textual output.

There are also several opcodes that underline the simplicity of the generation system in use at this stage. Only statistics about the relationship between pairs of items (opcode and opcode or opcode and argument) are held, with no conditional probability data beyond this. I have used the assumption that if no statistics are held then then a distribution is held to be uniform. Thus we see that the opcode **IMMEDIATEBYTEARG1ARG1=0** is recommended, which is reasonable enough. However the program has assumed that all opcodes that follow **IMMEDIATEBYTE** (from which this opcode was created) will follow the new opcode, in proportion to the frequency of the two opcodes. This includes the opcode for subtraction, **MINUS**. However whilst the compiler generates code to subtract various constants, zero is not such a constant, and so **MINUS** will never follow the pushing of constant zero. Thus we may suspect that the suggestion of opcode 190, **IMMEDIATEBYTEARG1ARG1=0MINUS** will not be particularly useful. This is clearly a limitation in the existing system for generating opcodes.

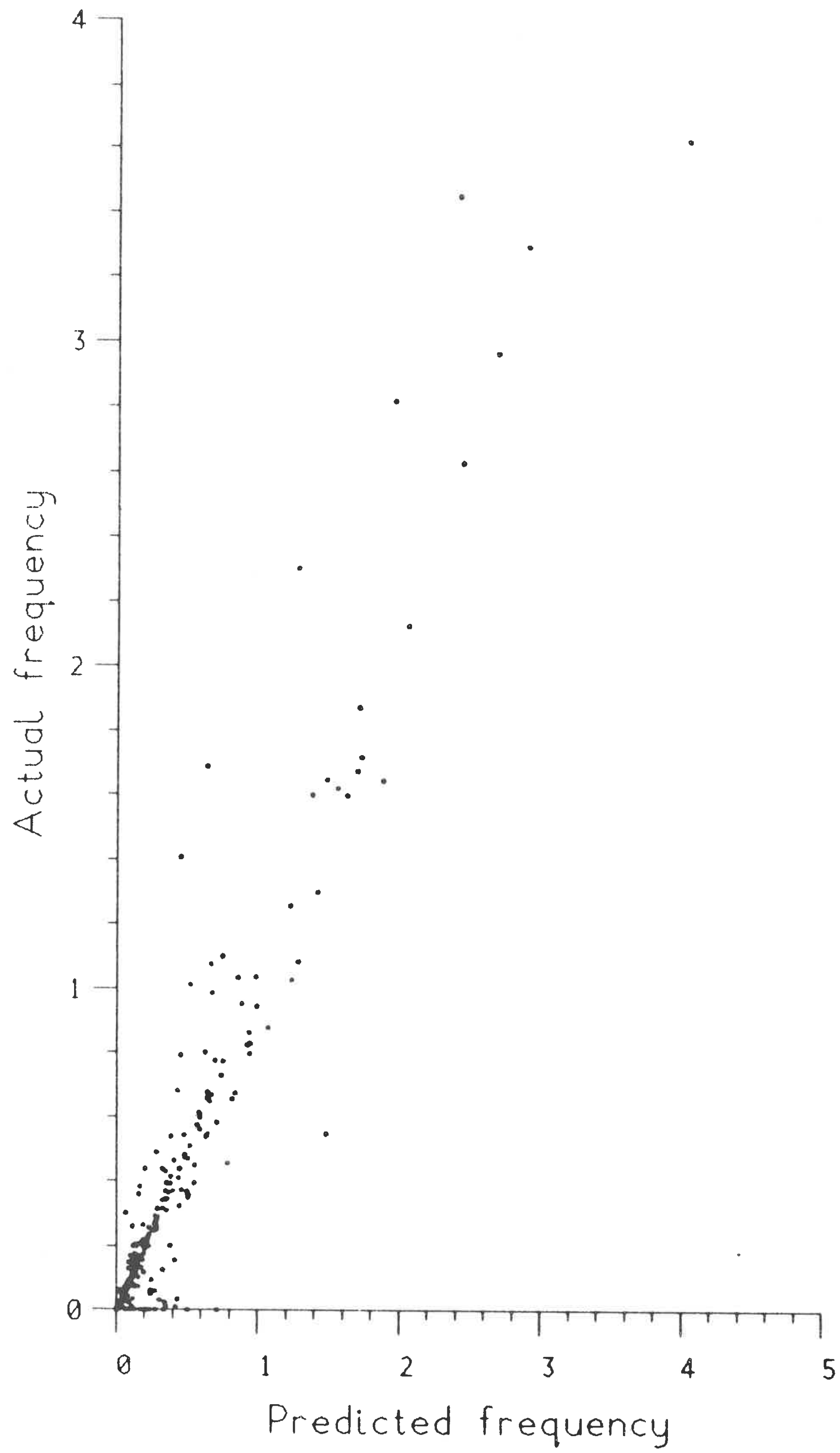
Another failing is that in the end we are left with some opcodes which we expect never to occur. Their entire occurrence has been subsumed into other opcodes. It would be relatively simple to cause these opcodes to be deleted, but it may be necessaary to provide a manual override of such a facility, to avoid deletion of opcodes such as **UNKNOWN**, which are there for debugging or similar purposes.

2.5. Testing the New Instruction Set

It is a fairly simple matter to test the new instruction set, by use of a peephole optimiser to apply it to the existing compiled code. Appendix II shows the effect of applying such optimisation, opcode by opcode. In general the reduction obtained is reasonably close to that predicted, and this is summarised by Graph 2.1., which shows the cumulative effect, predicted and actual, of the opcodes as they are applied. The predicted reduction in code size was to 28.16% of original, the reduction obtained was 28.55% of original. Graph 2.2. is a scatter diagram of actual against predicted opcode frequency. This shows that although there is general agreement (the product moment correlation coefficient is 0.94), there are several cases where opcodes turn out to be quite a lot more or less frequent than predicted. For example Table 2.3. shows the 10 most frequenct predicted and actual opcodes. This shows that even in the commonest opcodes, assumptions about the data may be wrong. Many of the other discrepancies are in the latter part of the generation, where the errors are very small in real terms (savings of 0.03%, rather than 0.06%, a few hundred bytes of the sample data). It is however worth considering the 14 opcodes which turned out to give give no saving at all. Although the reasons for some of them being unused are far from



Graph 2.1. Cumulative reduction in code size v. Opcode number



Graph 2.2. Actual against Predicted Instruction Frequency

Predicted		*Actual*	
Instruction	*Freq.*	*Instruction*	*Freq.*
GLOBALPUSHBYTEARG1CALLBYTEARG1	4.03%	GLOBALPUSHBYTEARG1CALLBYTEARG1	3.62%
GLOBALPUSHBYTEARG1	2.89%	JUMPBYTEARG1	3.45%
LOCALPUSHBYTEARG1	2.68%	GLOBALPUSHBYTEARG1	3.29%
IMMEDIATEBYTEARG1	2.43%	LOCALPUSHBYTEARG1	2.96%
JUMPBYTEARG1	2.41%	GLOBALPUSHHALFWORDARG1CALLBYTEARG1	2.82%
LOCALPOPBYTEARG1	2.05%	IMMEDIATEBYTEARG1	2.63%
GLOBALPUSHHALFWORDARG1CALLBYTEARG1	1.95%	LOCALPOPBYTEARG1LOCALPUSHBYTEARG1	2.30%
GLOBALPOPBYTEARG1	1.88%	LOCALPOPBYTEARG1	2.12%
IMMEDIATEHALFWORDARG1	1.72%	IMMEDIATEBYTEARG1ARG1=0	1.87%
IMMEDIATEBYTEARG1ARG1=0	1.71%	IMMEDIATEHALFWORDARG1	1.72%

Table 2.3. The Ten Commonest Instructions

obvious there are several cases where errors have crept in because of data dependances. There is **IMMEDIATEBYTEARG1ARG1=0MINUS**. As mentioned earlier constants are often subtracted, but not all equally often (the assumption made). Subtracting 0 is not useful. Similarly **GLOBALPUSHBYTEARG1ARG1=73** is not used, since this opcode is always followed by **CALL**, since it is the global number of **writes**, and has thus already been optimised out by **GLOBALPUSHBYTEARG1CALLBYTEARG1**.

Table 2.4. Shows a comparison of expected code sizes for the machines considered earlier against this new instruction set (*cf.* Table 1.2.). These figures were obtained by running the sample programs used here through the standard BCPL compiler and measuring the amount of OCODE produced. This would tend to indicate that the new method of design has something to offer. The excessive size of the canonical instruction set is due to the less than efficient representation used (one word for each argument).

Target Machine	*Relative Code Size*
Canonical Instruction Set	1.24
IBM 370	1.00
MC68000	0.84
OCODE	0.65
CINTCODE	0.52
New Instruction Set	0.35

Table 2.4. Static Code Size for Compiled BCPL

3. Future Work

There is scope for extending this work in a number of areas. The current system is very much the simplest possible example of the approach to be taken.

3.1. Limitations of the Current System

At present this system suffers from a number of deficiencies. At present the compiler for the canonical instruction set has not been properly tested, and almost certainly generates some incorrrect code. It is in any case a very simple compiler, and there is little doubt that at this stage better results in compiled code could come as much from the compiler as the instruction set.

The system also suffers from having insufficient design rules of sufficient power, since each design rule must be hand coded. A full blown system would probably need to consider many design rules, including possibly design rules for design rules, to deal with large numbers of very similar design rules. At present the system cannot cope correctly with opcodes such as **SWITCH** which take a variable number of opcodes, depending on the value of one of the arguments. It may also be necessary to have some way of specifying changes needed in the compiler (e.g. allocation of frequently used local variables to the smallest stack offsets).

There are clearly problems with the use of statistical data. The data is a summary of information about the compiled code for the canonical instruction set. One improvement would be to hold the compiled code in memory (it is only 500K) and do peephole optimisation each time an opcode is created, generating new statistics from the optimised code. This would certainly mean that the predicted and actual instruction sets should behave identically, but at considerable cost in execution time. This still does not overcome the fact that opcodes that are no use may be generated, but

it would be known that they were not of any value. A refinement would be to consider more than one possible contender at each attempt, chosing that which works best in reality.

Another aspect ignored by the current system is that it does not take into account that many arguments are program counter relative. As the program shrinks, so the arguments to these opcodes will shrink. Thus in achivieving a four-fold reduction in static code size, we would expect that many more **JUMP** opcodes would be able to become **JUMPBYTE** opcodes. This should be a fairly straightforward addition to the program.

In all these cases however the algorithm looks at only the next opcode to create. It is necessary to consider the possibility of using lookahead, to judge the benefit of some possible design rules. A rule that eliminates redundant opcodes, is of itself no benefit, but allows more opcodes to be created for example. Given that the current simple algorithm takes between 5 and 10 minutes CPU on a High Level Hardware ORION, such lookahead algorithms would be far from trivial to implement.

A more advanced version of the system is considered in section 3.3.

3.2. Application to Other Language Systems

The system as it stands is suited to any architecture that has single byte opcodes with variable number of arguments. It has also been applied as a design tool to the abstract machine used by the POLY programming language [Matthews85]. This has an existing byte stream instruction set of about 24 opcodes that support a 16 bit byte addressed machine. This existing instruction set was taken as the canonical instruction set, and a new instruction set of 256 opcodes generated. This instruction set gave a predicted reduction in static code size to about 45% of original. This is a simple demonstration of the power of a system such as this.

It should be perfectly feasible to design a system for multiple languages. The present design for BCPL gets nearly 90% of its improvement from the first 90 or so opcodes. Even if 256 opcodes were to prove inadequate, there is no reason why less common opcodes cannot be relegated to multiple byte opcodes.

3.3. A More General System

Section 3.1. pointed out some of the obvious limitations of the current system as applied to BCPL. The big problem is really the cost of

implementing each design rule as part of the code. The major priority must be a system that permits writing of design rules in a simple language, which can then be compiled or interpreted. A feasible initial system would probably restrict itself to the case where each opcode consists of an opcode byte with a number of argument bytes.

With the strain taken off the implementation of design rules in longhand more effort can be put into the algortihm for selecting new opcodes. The current system of summarising statistical data is clearly inadequate. A more feasible system might use very simple statistics to put up a number of canidates for best opcode to generate next, and then use peephole examination of the code to make a final decision. In all probability there would be no need to examine all the code at this stage. Peephole substitution of a new opcode, and generation of new statistics would be perfectly feasible, and a fixed cost (however many possibilities you consider, only one opcode is ever chosen per cycle, and there are a limited number of cycles, usually 256).

If this can be implemented efficiently it then becomes reasonable to consider some form of lookahead, although presumably with fair pruning of the search tree.

As such a system stands it would be possible to come up with an instruction set for each system as required. A future possibility to consider is that it might be possible given microcode for the canonical instruction set, and suitable information in the design rules, to generate the microcode for the new instruction set. In such an environment changing machine architecure to suit the current workload becomes a possibility. This should offer some novel problems for operating system designers.

4. Bibliography

Acorn85 — *Acorn RISC Machine CPU Software Manual.* Acorn Computers Ltd. 1985.

DEC81 — *VAX Architecture Handbook.* Digital Equipment Corp. 1985, 213-217.

HLH85 — *Microcoding the ORION.* High Level Hardware 1985.

Knuth71 — D.E. Knuth, *An Empirical Study of FORTREAN Programs.* Softw. Pract. Exper., 1, 1971, 105 - 133.

Matthews85 — *Poly Manual.* Cambridge University Computer Laboratory Technical Report No. 63, 1985.†

Patterson81 — D.A. Patterson and C.H. Séquin, *RISC I: A Reduced Instruction Set VLSI Computer.* Proc. Eighth Int'l Symp. Comp. Arch., May 1981, 443 - 457.

Richards79 — M.Richards, A.R.Aylward, P. Bond, R.D. Evans and B.J. Knight, *Tripos - A Portable Operating System for Mini-Computers.* Softw. Pract. Exper., 9, 1979, 513 - 529.

Richards82 — J. Richards and C. Jobson, *BCPL for the BBC microcomputer.* Acornsoft Ltd. 1983, 353 - 361, 381 - 386.

Sweet82 — R.E. Sweet and J.G. Sandman, *Empirical Analysis of the MESA Instruction Set.* ACM Symp. on Arch. Support for Prog. Lang. & Op. Sys., March 1982, 235 - 243.

Tanenbaum78 A.S. Tannenbaum, *Implications of Structured Programming for Machine Architecture*. Comm. ACM, 21 (3), March 1978, 237 - 246.

Weicker84 R.P. Weicker, *Dhrystone: A Synthetic Systems Programming Benchmark*. Comm. ACM, 27 (10), Sept 1984, 1013 - 1030.

Appendix I. Instruction Set Generation

This is the output from the **ISGEN** instruction set generation program, when applied to the BCPL canonical 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%, IFWHILEBYTEARG1
 55 redn 96.74%, total 54.10%, GLOBALPOPBYTEARG1
 56 redn 96.70%, total 52.31%, JUMPBYTEARG1
 57 redn 97.27%, total 50.88%, GLOBALPUSHBYTEARG1CALLBYTEARG1
 58 redn 97.19%, total 49.46%, IMMEDIATEBYTEARG1ARG1=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%, LOCALPUSHBYTEARG1ARG1=0
 63 redn 98.50%, total 44.84%, GLOBALPUSHHALFWORDARG1
```

```
 64 redn 98.60%, total 44.21%, IMMEDIATEBYTEARG1ARG1=1
 65 redn 98.68%, total 43.63%, PLUSPUSH
 66 redn 98.70%, total 43.06%, UNLESSUNTILBYTEARG1
 67 redn 98.74%, total 42.52%, LOCALPOPBYTEARG1LOCALPUSHBYTEARG1
 68 redn 98.86%, total 42.03%, ENDFORARG2=1
 69 redn 99.04%, total 41.63%, LOCALPUSHBYTEARG1ARG1=1
 70 redn 99.05%, total 41.23%, FORBYTEARG1
 71 redn 99.04%, total 40.83%, FORBYTEARG1BYTEARG2
 72 redn 99.10%, total 40.47%, EQIFWHILEBYTEARG1
 73 redn 99.10%, total 40.10%, ENDFORARG2=1BYTEARG1
 74 redn 99.10%, total 39.74%, FORBYTEARG1BYTEARG2BYTEARG3
 75 redn 99.15%, total 39.41%, STATICPUSHBYTEARG1CALLBYTEARG1
 76 redn 99.16%, total 39.08%, ENDFORARG2=1BYTEARG1BYTEARG3
 77 redn 99.17%, total 38.75%, IMMEDIATEHALFWORDARG1
 78 redn 99.18%, total 38.43%, LOCALPUSHBYTEARG1ARG1=2
 79 redn 99.18%, total 38.12%, STATICPUSHLEFTHALFWORDARG1BYTEARG1
 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%, LOCALPOPBYTEARG1ARG1=0
 85 redn 99.39%, total 36.58%, IMMEDIATEBYTEARG1ARG1=2
 86 redn 99.42%, total 36.37%, LOCALPUSHBYTEARG1ARG1=3
 87 redn 99.47%, total 36.18%, LOCALPOPBYTEARG1ARG1=1
 88 redn 99.50%, total 36.00%, GLOBALPUSHHALFWORDARG1CALLBYTEARG1
 89 redn 99.52%, total 35.82%, GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=0
 90 redn 99.54%, total 35.66%, LOCALPUSHBYTEARG1ARG1=4
 91 redn 99.54%, total 35.50%, LOCALPOPBYTEARG1ARG1=2
 92 redn 99.58%, total 35.34%, GLOBALPUSHBYTEARG1IMMEDIATEBYTEARG1ARG1=0
 93 redn 99.59%, total 35.20%, IMMEDIATEBYTEARG1ARG1=3
 94 redn 99.61%, total 35.06%, SWITCHBYTEARG2
 95 redn 99.62%, total 34.93%, LOCALPUSHBYTEARG1ARG1=5
 96 redn 99.62%, total 34.79%, GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=1
 97 redn 99.62%, total 34.66%, GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=2
 98 redn 99.63%, total 34.53%, GLOBALPUSHBYTEARG1IMMEDIATEBYTEARG1
 99 redn 99.63%, total 34.40%, GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=3
100 redn 99.64%, total 34.28%, LOCALPOPBYTEARG1ARG1=3
101 redn 99.67%, total 34.17%, LOCALPUSHBYTEARG1IMMEDIATEBYTEARG1ARG1=0
102 redn 99.68%, total 34.06%, EQUNLESSUNTILBYTEARG1
103 redn 99.70%, total 33.96%, LOCALPOPBYTEARG1ARG1=4
104 redn 99.71%, total 33.86%, IMMEDIATEBYTEARG1ARG1=10
```

```
105 redn 99.71%, total 33.76%, LOCALPUSHBYTEARG1ARG1=6
106 redn 99.71%, total 33.66%, LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG2=1
107 redn 99.72%, total 33.57%, IMMEDIATEBYTEARG1ARG1=0EQIFWHILEBYTEARG1
108 redn 99.73%, total 33.47%, SWITCHBYTEARG2HALFWORDARG1
109 redn 99.73%, total 33.38%, IMMEDIATEBYTEARG1ARG1=4
110 redn 99.73%, total 33.29%, GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=4
111 redn 99.73%, total 33.21%, EQLOGOR
112 redn 99.74%, total 33.12%, NEIFWHILEBYTEARG1
113 redn 99.74%, total 33.03%, LOGANDIFWHILEBYTEARG1
114 redn 99.74%, total 32.95%, STATICPUSHHALFWORDARG1
115 redn 99.75%, total 32.86%, REPEATHALFWORDARG1
116 redn 99.75%, total 32.78%, GLOBALPUSHBYTEARG1GLOBALPUSHBYTEARG1
117 redn 99.75%, total 32.70%, LOCALPUSHBYTEARG1ARG1=0IMMEDIATEBYTEARG1ARG1=0
118 redn 99.76%, total 32.62%, LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG2=2
119 redn 99.76%, total 32.55%, GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=5
120 redn 99.77%, total 32.47%, LOCALPOPBYTEARG1ARG1=5
121 redn 99.77%, total 32.40%, IMMEDIATEBYTEARG1EQIFWHILEBYTEARG1
122 redn 99.78%, total 32.33%, IMMEDIATEBYTEARG1ARG1=0GLOBALPOPBYTEARG1
123 redn 99.78%, total 32.25%, JUMPBYTEARG1RETURN
124 redn 99.79%, total 32.19%, GLOBALPOPHALFWORDARG1
125 redn 99.79%, total 32.12%, IMMEDIATEBYTEARG1ARG1=0PLUSPUSH
126 redn 99.79%, total 32.05%, LOCALPUSHBYTEARG1IMMEDIATEBYTEARG1
127 redn 99.80%, total 31.99%, STATICPUSHBYTEARG1GOTO
128 redn 99.80%, total 31.92%, GLOBALPUSHBYTEARG1PLUSPUSH
129 redn 99.81%, total 31.86%, GLOBALPUSHBYTEARG1IMMEDIATEBYTEARG1ARG1=1
130 redn 99.81%, total 31.80%, STATICPOPBYTEARG1
131 redn 99.81%, total 31.74%, LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=0
132 redn 99.82%, total 31.69%, LOCALPUSHBYTEARG1ARG1=0IMMEDIATEBYTEARG1
133 redn 99.82%, total 31.63%, LOCALPOPBYTEARG1ARG1=6
134 redn 99.82%, total 31.57%, GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=6
135 redn 99.83%, total 31.52%, IFWHILEHALFWORDARG1
136 redn 99.83%, total 31.47%, LOCALPUSHBYTEARG1ARG1=8
137 redn 99.83%, total 31.41%, LOCALPUSHBYTEARG1ARG1=7
138 redn 99.83%, total 31.36%, GLOBALPOPBYTEARG1IMMEDIATEBYTEARG1ARG1=0
139 redn 99.84%, total 31.31%, LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=1
140 redn 99.84%, total 31.26%, GRIFWHILEBYTEARG1
141 redn 99.84%, total 31.21%, GLOBALPUSHBYTEARG1GLOBALPUSHBYTEARG1CALLBYTEARG1
142 redn 99.85%, total 31.16%, IMMEDIATEBYTEARG1PLUSPUSH
143 redn 99.85%, total 31.12%, JUMPBYTEARG1IMMEDIATEBYTEARG1ARG1=0
144 redn 99.85%, total 31.07%, IMMEDIATEBYTEARG1GLOBALPOPBYTEARG1
145 redn 99.85%, total 31.02%, LOCALPUSHBYTEARG1ARG1=0PLUSPUSH
```

```
146 redn 99.85%, total 30.98%, IMMEDIATEBYTEARG1ARG1=1EQIFWHILEBYTEARG1
147 redn 99.85%, total 30.93%, PLUSGLOBALPOPBYTEARG1
148 redn 99.85%, total 30.89%, LOCALPUSHBYTEARG1PLUSPUSH
149 redn 99.85%, total 30.84%, GLOBALPUSHLEFTBYTEARG1
150 redn 99.86%, total 30.80%, LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=2
151 redn 99.87%, total 30.76%, STATICPUSHBYTEARG1CALLBYTEARG1ARG2=0
152 redn 99.87%, total 30.72%, STATICPUSHLEFTHALFWORDARG1BYTEARG1GLOBALPUSHBYTEARG1
153 redn 99.87%, total 30.68%, LOCALPOPBYTEARG1FORBYTEARG1BYTEARG2BYTEARG3
154 redn 99.87%, total 30.64%, LOCALPOPBYTEARG1IMMEDIATEBYTEARG1ARG1=0
155 redn 99.87%, total 30.60%, IMMEDIATEBYTEARG1ARG1=5
156 redn 99.87%, total 30.56%, GLOBALPUSHBYTEARG1CALLBYTEARG1RETURN
157 redn 99.88%, total 30.52%, IMMEDIATEBYTEARG1ARG1=8
158 redn 99.88%, total 30.49%, LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=0
159 redn 99.88%, total 30.45%, IMMEDIATEBYTEARG1ARG1=7
160 redn 99.88%, total 30.41%, LOCALPUSHBYTEARG1ARG1=1IMMEDIATEBYTEARG1ARG1=0
161 redn 99.88%, total 30.38%, GLOBALPUSHBYTEARG1GLOBALPOPBYTEARG1
162 redn 99.88%, total 30.34%, IMMEDIATEBYTEARG1ARG1=1PLUSPUSH
163 redn 99.88%, total 30.31%, GLOBALPUSHBYTEARG1CALLBYTEARG1JUMPBYTEARG1
164 redn 99.88%, total 30.27%, LOCALPUSHBYTEARG1ARG1=9
165 redn 99.89%, total 30.24%, LOCALPUSHBYTEARG1ARG1=11
166 redn 99.89%, total 30.20%, IMMEDIATEBYTEARG1ARG1=1GLOBALPOPBYTEARG1
167 redn 99.89%, total 30.17%, IMMEDIATEBYTEARG1ARG1=6
168 redn 99.89%, total 30.14%, GLOBALPUSHBYTEARG1PLUS
169 redn 99.89%, total 30.10%, ENDFORBYTEARG1
170 redn 99.89%, total 30.07%, LSIFWHILEBYTEARG1
171 redn 99.89%, total 30.04%, IMMEDIATEBYTEARG1ARG1=0EQ
172 redn 99.89%, total 30.01%, STATICPUSHBYTEARG1CALLBYTEARG1ARG1=0
173 redn 99.89%, total 29.97%, IMMEDIATEBYTEARG1ARG1=32
174 redn 99.89%, total 29.94%, GLOBALPUSHBYTEARG1IFWHILEBYTEARG1
175 redn 99.89%, total 29.91%, UNLESSUNTILHALFWORDARG1
176 redn 99.89%, total 29.88%, LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=3
177 redn 99.89%, total 29.84%, REPEATUNTILBYTEARG1
178 redn 99.89%, total 29.81%, IMMEDIATEBYTEARG1ARG1=20
179 redn 99.89%, total 29.78%, STATICPUSHLEFTHALFWORDARG1BYTEARG1GLOBALPUSHBYTEARG1CALLBYTEARG1
180 redn 99.90%, total 29.75%, STATICPUSHHALFWORDARG1CALLBYTEARG1
181 redn 99.90%, total 29.72%, EQLOGORIFWHILEBYTEARG1
182 redn 99.90%, total 29.69%, IMMEDIATEBYTEARG1ARG1=0PLUS
183 redn 99.90%, total 29.66%, ENDFORBYTEARG1BYTEARG3
184 redn 99.90%, total 29.63%, LOCALPOPBYTEARG1ARG1=7
185 redn 99.90%, total 29.60%, LOCALPUSHBYTEARG1ARG1=0IMMEDIATEBYTEARG1ARG1=1
186 redn 99.91%, total 29.57%, GLOBALPUSHHALFWORDARG1CALLBYTEARG1ARG1=256
```

```
187 redn 99.91%, total 29.55%, STATICPUSHBYTEARG1CALLBYTEARG1ARG2=2
188 redn 99.91%, total 29.52%, STATICPUSHBYTEARG1CALLBYTEARG1ARG2=1
189 redn 99.91%, total 29.49%, STATICPUSHBYTEARG1CALLBYTEARG1ARG2=3
190 redn 99.91%, total 29.46%, IMMEDIATEBYTEARG1ARG1=0MINUS
191 redn 99.91%, total 29.44%, GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=7
192 redn 99.91%, total 29.41%, LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=4
193 redn 99.91%, total 29.39%, LELOGAND
194 redn 99.91%, total 29.36%, SWITCHBYTEARG2HALFWORDARG1BYTEARG1
195 redn 99.91%, total 29.33%, LOCALPUSHBYTEARG1IMMEDIATEBYTEARG1ARG1=1
196 redn 99.91%, total 29.31%, STATICPUSHLEFTHALFWORDARG1BYTEARG1JUMPBYTEARG1
197 redn 99.92%, total 29.28%, GLOBALPUSHHALFWORDARG1ARG1=256
198 redn 99.92%, total 29.26%, LOCALPUSHBYTEARG1ARG1=0PLUS
199 redn 99.92%, total 29.23%, IMMEDIATEBYTEARG1ARG1=48
200 redn 99.92%, total 29.21%, GLOBALPUSHBYTEARG1PLUSPOP
201 redn 99.92%, total 29.18%, LOCALPUSHBYTEARG1ARG1=1PLUSPUSH
202 redn 99.92%, total 29.16%, LOCALPUSHBYTEARG1ARG1=2IMMEDIATEBYTEARG1ARG1=0
203 redn 99.92%, total 29.14%, FORBYTEARG1BYTEARG2HALFWORDARG3
204 redn 99.92%, total 29.11%, GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=9
205 redn 99.92%, total 29.09%, IMMEDIATEBYTEARG1ARG1=9
206 redn 99.92%, total 29.07%, LOCALPUSHBYTEARG1PLUS
207 redn 99.92%, total 29.04%, LOCALPOPBYTEARG1ARG1=8
208 redn 99.92%, total 29.02%, GLOBALPUSHHALFWORDARG1CALLBYTEARG1ARG1=259
209 redn 99.92%, total 29.00%, IMMEDIATEBYTEARG1ARG1=0JUMPBYTEARG1
210 redn 99.93%, total 28.98%, LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG2=3
211 redn 99.93%, total 28.96%, GLOBALPOPBYTEARG1RETURN
212 redn 99.93%, total 28.94%, LOCALPUSHBYTEARG1ARG1=13
213 redn 99.93%, total 28.91%, ENDFORARG2=1BYTEARG1HALFWORDARG3
214 redn 99.93%, total 28.89%, LOGORIFWHILEBYTEARG1
215 redn 99.93%, total 28.87%, IMMEDIATEBYTEARG1ARG1=0PLUSPOP
216 redn 99.93%, total 28.85%, GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=10
217 redn 99.93%, total 28.83%, GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=8
218 redn 99.93%, total 28.81%, GLOBALPUSHBYTEARG1IMMEDIATEBYTEARG1ARG1=2
219 redn 99.93%, total 28.79%, REPEATWHILEBYTEARG1
220 redn 99.93%, total 28.77%, LOCALPUSHBYTEARG1ARG1=10
221 redn 99.93%, total 28.75%, JUMPBYTEARG1STATICPUSHLEFTHALFWORDARG1BYTEARG1
222 redn 99.93%, total 28.73%, LOCALPOPBYTEARG1ARG1=0FORBYTEARG1BYTEARG2BYTEARG3
223 redn 99.93%, total 28.71%, GLOBALPUSHHALFWORDARG1ARG1=259
224 redn 99.93%, total 28.70%, ENDFORBYTEARG1BYTEARG3ARG2=4294967295
225 redn 99.93%, total 28.68%, LOCALPUSHBYTEARG1ARG1=2PLUSPUSH
226 redn 99.93%, total 28.66%, IMMEDIATEBYTEARG1ARG1=0EQUNLESSUNTILBYTEARG1
227 redn 99.93%, total 28.64%, IMMEDIATEBYTEARG1ARG1=1EQ
```

```
228 redn 99.93%, total 28.62%, GLOBALPUSHBYTEARG1ARG1=74
229 redn 99.93%, total 28.60%, GEIFWHILEBYTEARG1
230 redn 99.93%, total 28.58%, JUMPHALFWORDARG1RETURN
231 redn 99.93%, total 28.56%, STATICPUSHBYTEARG1CALLBYTEARG1ARG2=4
232 redn 99.93%, total 28.54%, GLOBALPUSHHALFWORDARG1CALLBYTEARG1ARG2=0
233 redn 99.94%, total 28.53%, GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=12
234 redn 99.94%, total 28.51%, LOCALPUSHBYTEARG1ARG1=12
235 redn 99.94%, total 28.49%, LOCALPOPBYTEARG1ARG1=9
236 redn 99.94%, total 28.47%, GLOBALPOPBYTEARG1ARG1=10
237 redn 99.94%, total 28.45%, GLOBALPUSHBYTEARG1IMMEDIATEARG1=4294967295
238 redn 99.94%, total 28.44%, JUMPBYTEARG1LOCALPUSHBYTEARG1ARG1=0
239 redn 99.94%, total 28.42%, POPBYTEENDFORARG2=1BYTEARG1BYTEARG3
240 redn 99.94%, total 28.40%, LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG2=4
241 redn 99.94%, total 28.38%, IMMEDIATEBYTEARG1ARG1=1PLUS
242 redn 99.94%, total 28.37%, LOCALPUSHBYTEARG1ARG1=0PLUSPOP
243 redn 99.94%, total 28.35%, NELOGAND
244 redn 99.94%, total 28.33%, GLOBALPUSHBYTEARG1ARG1=73
245 redn 99.94%, total 28.32%, IMMEDIATEBYTEARG1EQ
246 redn 99.94%, total 28.30%, STATICPUSHBYTEARG1CALLBYTEARG1ARG2=5
247 redn 99.94%, total 28.28%, LELOGANDIFWHILEBYTEARG1
248 redn 99.94%, total 28.27%, IMMEDIATEBYTEARG1ARG1=2EQIFWHILEBYTEARG1
249 redn 99.94%, total 28.25%, LOCALPOPBYTEARG1ARG1=1FORBYTEARG1BYTEARG2BYTEARG3
250 redn 99.94%, total 28.23%, LOCALPUSHLEFTBYTEARG1ARG1=1
251 redn 99.94%, total 28.22%, MINUSGLOBALPOPBYTEARG1
252 redn 99.94%, total 28.20%, JUMPBYTEARG1LOCALPUSHBYTEARG1
253 redn 99.94%, total 28.19%, IMMEDIATEBYTEARG1ARG1=0NEIFWHILEBYTEARG1
254 redn 99.94%, total 28.17%, LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG2=1ARG1=0
255 redn 99.94%, total 28.16%, IMMEDIATEARG1=4294967295EQIFWHILEBYTEARG1

275312 bytes of store used
Instruction set generated successfully
```

This is the suggested new instruction set.

Freq.	*Instruction*
4.0333%	GLOBALPUSHBYTEARG1CALLBYTEARG1
2.8937%	GLOBALPUSHBYTEARG1
2.6846%	LOCALPUSHBYTEARG1

2.4342%	IMMEDIATEBYTEARG1
2.4115%	JUMPBYTEARG1
2.0529%	LOCALPOPBYTEARG1
1.9548%	GLOBALPUSHHALFWORDARG1CALLBYTEARG1
1.8808%	GLOBALPOPBYTEARG1
1.7222%	IMMEDIATEHALFWORDARG1
1.7079%	IMMEDIATEBYTEARG1ARG1=0
1.6944%	STATICPUSHLEFTHALFWORDARG1
1.6247%	LOCALPUSHBYTEARG1ARG1=0
1.5578%	STATICPUSHBYTEARG1CALLBYTEARG1
1.4789%	JUMPHALFWORDARG1
1.4767%	GLOBALPUSHHALFWORDARG1
1.4142%	STATICPUSHLEFTHALFWORDARG1BYTEARG1
1.3807%	GLOBALPUSHBYTEARG1IMMEDIATEBYTEARG1
1.2769%	IMMEDIATEBYTEARG1ARG1=1
1.2719%	LOCALPOPBYTEARG1LOCALPUSHBYTEARG1
1.2335%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=0
1.2200%	LOCALPUSHBYTEARG1ARG1=1
1.0684%	GLOBALPUSHBYTEARG1IMMEDIATEBYTEARG1ARG1=0
0.9881%	ENDFORARG2=1BYTEARG1BYTEARG3
0.9767%	LOCALPUSHBYTEARG1ARG1=2
0.9376%	IFWHILEBYTEARG1
0.9347%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=1
0.9304%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=2
0.9191%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=3
0.8771%	GLOBALPUSHBYTEARG1GLOBALPUSHBYTEARG1
0.8536%	EQIFWHILEBYTEARG1
0.8344%	PLUSPUSH
0.8138%	LOCALPUSHBYTEARG1IMMEDIATEBYTEARG1ARG1=0
0.7853%	IMMEDIATEBYTEARG1EQIFWHILEBYTEARG1
0.7483%	LOCALPUSHBYTEARG1ARG1=3
0.7412%	LOCALPOPBYTEARG1ARG1=0
0.7377%	PLUS
0.7113%	LOCALPOPBYTEARG1FORBYTEARG1BYTEARG2BYTEARG3
0.7042%	LOCALPUSHBYTEARG1IMMEDIATEBYTEARG1
0.6914%	GLOBALPUSHBYTEARG1GLOBALPUSHBYTEARG1CALLBYTEARG1
0.6701%	IMMEDIATEBYTEARG1ARG1=0EQIFWHILEBYTEARG1
0.6658%	IMMEDIATEBYTEARG1ARG1=2
0.6637%	RETURN
0.6559%	REPEATBYTEARG1
0.6416%	EQUNLESSUNTILBYTEARG1

0.6395%	IMMEDIATEARG1=4294967295
0.6395%	PLUSPOP
0.6345%	FORBYTEARG1BYTEARG2BYTEARG3
0.6303%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=4
0.6217%	LOCALPOPBYTEARG1ARG1=1
0.5876%	LOCALPUSHBYTEARG1ARG1=4
0.5861%	LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG2=1
0.5812%	LOCALPOPBYTEARG1ARG1=2
0.5662%	UNLESSUNTILBYTEARG1
0.5506%	LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG2=2
0.5477%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=5
0.5164%	IMMEDIATEBYTEARG1ARG1=3
0.5150%	IMMEDIATEBYTEARG1ARG1=0GLOBALPOPBYTEARG1
0.5079%	NEIFWHILEBYTEARG1
0.5036%	LOGANDIFWHILEBYTEARG1
0.5036%	JUMPBYTEARG1RETURN
0.5008%	GLOBALPUSHBYTEARG1CALLBYTEARG1JUMPBYTEARG1
0.4972%	IMMEDIATEBYTEARG1GLOBALPOPBYTEARG1
0.4794%	LOCALPUSHLEFTBYTEARG1
0.4787%	LOCALPUSHBYTEARG1ARG1=5
0.4759%	MINUS
0.4581%	STATICPUSHBYTEARG1GOTO
0.4510%	GLOBALPUSHBYTEARG1PLUSPUSH
0.4467%	STATICPUSHLEFTHALFWORDARG1BYTEARG1GLOBALPUSHBYTEARG1CALLBYTEARG1
0.4460%	PUSHBYTE
0.4439%	REPEATHALFWORDARG1
0.4439%	STATICPUSHHALFWORDARG1CALLBYTEARG1
0.4361%	LOCALPOPBYTEARG1ARG1=3
0.4289%	STATICPUSHLEFTHALFWORDARG1BYTEARG1GLOBALPUSHBYTEARG1
0.4282%	GLOBALPUSHBYTEARG1IMMEDIATEBYTEARG1ARG1=1
0.4211%	LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=0
0.4097%	GLOBALPUSHBYTEARG1CALLBYTEARG1RETURN
0.4055%	LOCALPUSHBYTEARG1ARG1=0IMMEDIATEBYTEARG1
0.3955%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=6
0.3820%	EQ
0.3799%	GLOBALPUSHBYTEARG1GLOBALPOPBYTEARG1
0.3770%	STATICPUSHBYTEARG1
0.3770%	GLOBALPOPBYTEARG1IMMEDIATEBYTEARG1ARG1=0
0.3692%	GLOBALPOPHALFWORDARG1
0.3571%	LOCALPOPBYTEARG1ARG1=4
0.3542%	LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=1

0.3535%	IMMEDIATEBYTEARG1ARG1=10
0.3528%	LOCALPUSHBYTEARG1ARG1=6
0.3457%	GRIFWHILEBYTEARG1
0.3436%	LOGAND
0.3436%	GLOBALPUSHBYTEARG1IFWHILEBYTEARG1
0.3400%	IMMEDIATEBYTEARG1PLUSPUSH
0.3358%	JUMPBYTEARG1IMMEDIATEBYTEARG1ARG1=0
0.3258%	IMMEDIATEBYTEARG1ARG1=1EQIFWHILEBYTEARG1
0.3244%	IMMEDIATEBYTEARG1ARG1=4
0.3230%	PLUSGLOBALPOPBYTEARG1
0.3230%	LOCALPUSHBYTEARG1PLUSPUSH
0.3030%	LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=2
0.2902%	IFWHILEHALFWORDARG1
0.2888%	STATICPUSHBYTEARG1CALLBYTEARG1ARG2=0
0.2867%	LOCALPUSHBYTEARG1ARG1=0IMMEDIATEBYTEARG1ARG1=0
0.2845%	SWITCHBYTEARG2HALFWORDARG1
0.2810%	IMMEDIATE
0.2803%	LOCALPOPBYTEARG1IMMEDIATEBYTEARG1ARG1=0
0.2774%	SWITCHBYTEARG2HALFWORDARG1BYTEARG1
0.2774%	STATICPUSHLEFTHALFWORDARG1BYTEARG1JUMPBYTEARG1
0.2760%	LOCALPOPBYTEARG1ARG1=0FORBYTEARG1BYTEARG2BYTEARG3
0.2696%	PUSH
0.2689%	LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=0
0.2625%	LOCALPOPBYTEARG1ARG1=5
0.2433%	IMMEDIATEBYTEARG1ARG1=1GLOBALPOPBYTEARG1
0.2404%	GLOBALPUSHBYTEARG1PLUS
0.2383%	IMMEDIATEBYTEARG1ARG1=0PLUSPUSH
0.2319%	LSIFWHILEBYTEARG1
0.2305%	STATICPUSHBYTEARG1CALLBYTEARG1ARG1=0
0.2305%	LOCALPOPBYTEARG1ARG1=1FORBYTEARG1BYTEARG2BYTEARG3
0.2276%	LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=3
0.2205%	EQLOGORIFWHILEBYTEARG1
0.2134%	FORBYTEARG1BYTEARG2HALFWORDARG3
0.2098%	LE
0.2091%	JUMPBYTEARG1STATICPUSHLEFTHALFWORDARG1BYTEARG1
0.2049%	EQLOGOR
0.2013%	LOCALPOPBYTEARG1ARG1=6
0.2006%	JUMPHALFWORDARG1RETURN
0.1985%	GLOBALPUSHHALFWORDARG1CALLBYTEARG1ARG2=0
0.1963%	STATICPUSHBYTEARG1CALLBYTEARG1ARG2=2
0.1949%	STATICPUSHBYTEARG1CALLBYTEARG1ARG2=1

```
0.1921%   LOCALPUSHBYTEARG1ARG1=8
0.1906%   NE
0.1906%   STATICPUSHBYTEARG1CALLBYTEARG1ARG2=3
0.1892%   LOCALPUSHBYTEARG1ARG1=7
0.1878%   POPBYTEENDFORARG2=1BYTEARG1BYTEARG3
0.1864%   GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=7
0.1864%   LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=4
0.1849%   LOCALPUSHBYTEARG1IMMEDIATEBYTEARG1ARG1=1
0.1736%   GLOBALPUSHBYTEARG1PLUSPOP
0.1707%   UNLESSUNTILHALFWORDARG1
0.1693%   GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=9
0.1686%   JUMPBYTEARG1LOCALPUSHBYTEARG1
0.1643%   LOCALPUSHBYTEARG1ARG1=0PLUSPUSH
0.1622%   LOCALPUSHBYTEARG1PLUS
0.1551%   IMMEDIATEBYTEARG1ARG1=0JUMPBYTEARG1
0.1537%   LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG2=3
0.1537%   GLOBALPOPBYTEARG1RETURN
0.1480%   ENDFORARG2=1BYTEARG1HALFWORDARG3
0.1465%   LOGORIFWHILEBYTEARG1
0.1451%   GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=10
0.1437%   GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=8
0.1423%   STATICPOPBYTEARG1
0.1423%   GLOBALPUSHBYTEARG1IMMEDIATEBYTEARG1ARG1=2
0.1380%   POPBYTE
0.1380%   LOGOR
0.1373%   IMMEDIATEBYTEARG1ARG1=5
0.1352%   IMMEDIATEBYTEARG1ARG1=0EQUNLESSUNTILBYTEARG1
0.1344%   IMMEDIATEBYTEARG1ARG1=8
0.1337%   GEIFWHILEBYTEARG1
0.1323%   STATICPUSHBYTEARG1CALLBYTEARG1ARG2=4
0.1316%   IMMEDIATEBYTEARG1ARG1=7
0.1309%   GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=12
0.1295%   ENDFORBYTEARG1BYTEARG3
0.1288%   LOCALPUSHBYTEARG1ARG1=1IMMEDIATEBYTEARG1ARG1=0
0.1280%   MULT
0.1280%   GLOBALPUSHBYTEARG1IMMEDIATEARG1=4294967295
0.1280%   JUMPBYTEARG1LOCALPUSHBYTEARG1ARG1=0
0.1273%   NOT
0.1259%   IMMEDIATEBYTEARG1ARG1=1PLUSPUSH
0.1245%   LOCALPUSHBYTEARG1ARG1=9
0.1238%   LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG2=4
```

0.1224%	LOCALPUSHBYTEARG1ARG1=11
0.1202%	IMMEDIATEBYTEARG1ARG1=6
0.1181%	IMMEDIATEBYTEARG1EQ
0.1174%	STATICPUSHHALFWORDARG1
0.1159%	IMMEDIATEBYTEARG1ARG1=0EQ
0.1152%	STATICPUSHBYTEARG1CALLBYTEARG1ARG2=5
0.1152%	LELOGANDIFWHILEBYTEARG1
0.1152%	IMMEDIATEBYTEARG1ARG1=2EQIFWHILEBYTEARG1
0.1145%	IMMEDIATEBYTEARG1ARG1=32
0.1124%	IMMEDIATEBYTEARG1ARG1=20
0.1124%	MINUSGLOBALPOPBYTEARG1
0.1124%	IMMEDIATEBYTEARG1ARG1=0NEIFWHILEBYTEARG1
0.1110%	IMMEDIATEARG1=4294967295EQIFWHILEBYTEARG1
0.1103%	DIV
0.1081%	IMMEDIATEBYTEARG1ARG1=0PLUS
0.1067%	RSHIFT
0.1067%	GLOBALPUSHLEFTBYTEARG1
0.1024%	LOCALPOPBYTEARG1ARG1=7
0.1003%	LOCALPUSHBYTEARG1ARG1=0IMMEDIATEBYTEARG1ARG1=1
0.0996%	GLOBALPUSHHALFWORDARG1CALLBYTEARG1ARG1=256
0.0989%	POP
0.0953%	IMMEDIATEBYTEARG1ARG1=0MINUS
0.0932%	LELOGAND
0.0903%	GR
0.0882%	LOCALPUSHBYTEARG1ARG1=0PLUS
0.0875%	IMMEDIATEBYTEARG1ARG1=48
0.0868%	LOCALPUSHBYTEARG1ARG1=1PLUSPUSH
0.0868%	LOCALPUSHBYTEARG1ARG1=2IMMEDIATEBYTEARG1ARG1=0
0.0839%	IMMEDIATEBYTEARG1ARG1=9
0.0797%	LOCALPOPBYTEARG1ARG1=8
0.0797%	GLOBALPUSHHALFWORDARG1CALLBYTEARG1ARG1=259
0.0754%	REPEATUNTILBYTEARG1
0.0747%	LOCALPUSHBYTEARG1ARG1=13
0.0733%	IMMEDIATEBYTEARG1ARG1=0PLUSPOP
0.0718%	LSHIFT
0.0704%	QUERY
0.0704%	LOCALPUSHBYTEARG1ARG1=10
0.0683%	LOCALPUSHBYTEARG1ARG1=2PLUSPUSH
0.0676%	IMMEDIATEBYTEARG1ARG1=1EQ
0.0669%	GLOBALPUSHBYTEARG1ARG1=74
0.0647%	LS

```
0.0647%  LOCALPUSHBYTEARG1ARG1=12
0.0647%  LOCALPOPBYTEARG1ARG1=9
0.0647%  GLOBALPOPBYTEARG1ARG1=10
0.0619%  IMMEDIATEBYTEARG1ARG1=1PLUS
0.0605%  LOCALPUSHBYTEARG1ARG1=0PLUSPOP
0.0598%  NELOGAND
0.0590%  GLOBALPUSHBYTEARG1ARG1=73
0.0562%  LOCALPUSHLEFTBYTEARG1ARG1=1
0.0555%  LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG2=1ARG1=0
0.0512%  REM
0.0512%  ENDFORBYTEARG1BYTEARG3ARG2=4294967295
0.0491%  NEG
0.0484%  GE
0.0469%  REPEATWHILEBYTEARG1
0.0441%  GLOBALPUSHHALFWORDARG1ARG1=256
0.0398%  CALLBYTEARG1
0.0356%  ENDFORBYTEARG1
0.0341%  GLOBALPUSHHALFWORDARG1ARG1=259
0.0320%  REPEATUNTIL
0.0228%  GOTO
0.0213%  REPEATWHILE
0.0206%  ABS
0.0178%  STATICPOP
0.0107%  GLOBALPUSHLEFT
0.0092%  ENDFOR
0.0085%  FINISH
0.0064%  ENDFORARG2=1
0.0050%  NEQV
0.0043%  ENDFORARG2=1BYTEARG1
0.0014%  EQV
0.0000%  UNKNOWN
0.0000%  GLOBALPUSH
0.0000%  STATICPUSH
0.0000%  LOCALPUSH
0.0000%  STATICPUSHLEFT
0.0000%  LOCALPUSHLEFT
0.0000%  GLOBALPOP
0.0000%  LOCALPOP
0.0000%  CALL
0.0000%  FOR
0.0000%  JUMP
```

```
0.0000%  REPEAT
0.0000%  IFWHILE
0.0000%  UNLESSUNTIL
0.0000%  SWITCH
0.0000%  FORBYTEARG1
0.0000%  FORBYTEARG1BYTEARG2
0.0000%  SWITCHBYTEARG2
```

Appendix II. Peephole Optimisation

This is the listing generated by the peephole optimiser, showing the benefit gained from the various opcodes.

#	*Pred*	*Cumm*	*Actual*	*Cumm*	*Instruction*
0	100.00%	100.00%	100.00%	100.00%	UNKNOWN
1	100.00%	100.00%	100.00%	100.00%	GLOBALPUSH
2	100.00%	100.00%	100.00%	100.00%	STATICPUSH
3	100.00%	100.00%	100.00%	100.00%	LOCALPUSH
4	100.00%	100.00%	100.00%	100.00%	PUSH
5	100.00%	100.00%	100.00%	100.00%	PUSHBYTE
6	100.00%	100.00%	100.00%	100.00%	GLOBALPUSHLEFT
7	100.00%	100.00%	100.00%	100.00%	STATICPUSHLEFT
8	100.00%	100.00%	100.00%	100.00%	LOCALPUSHLEFT
9	100.00%	100.00%	100.00%	100.00%	IMMEDIATE
10	100.00%	100.00%	100.00%	100.00%	GLOBALPOP
11	100.00%	100.00%	100.00%	100.00%	STATICPOP
12	100.00%	100.00%	100.00%	100.00%	LOCALPOP
13	100.00%	100.00%	100.00%	100.00%	POP
14	100.00%	100.00%	100.00%	100.00%	POPBYTE
15	100.00%	100.00%	100.00%	100.00%	QUERY

16	100.00%	100.00%	100.00%	100.00%	NEG
17	100.00%	100.00%	100.00%	100.00%	NOT
18	100.00%	100.00%	100.00%	100.00%	ABS
19	100.00%	100.00%	100.00%	100.00%	MULT
20	100.00%	100.00%	100.00%	100.00%	DIV
21	100.00%	100.00%	100.00%	100.00%	REM
22	100.00%	100.00%	100.00%	100.00%	PLUS
23	100.00%	100.00%	100.00%	100.00%	MINUS
24	100.00%	100.00%	100.00%	100.00%	EQ
25	100.00%	100.00%	100.00%	100.00%	NE
26	100.00%	100.00%	100.00%	100.00%	LS
27	100.00%	100.00%	100.00%	100.00%	GR
28	100.00%	100.00%	100.00%	100.00%	LE
29	100.00%	100.00%	100.00%	100.00%	GE
30	100.00%	100.00%	100.00%	100.00%	LSHIFT
31	100.00%	100.00%	100.00%	100.00%	RSHIFT
32	100.00%	100.00%	100.00%	100.00%	LOGAND
33	100.00%	100.00%	100.00%	100.00%	LOGOR
34	100.00%	100.00%	100.00%	100.00%	EQV
35	100.00%	100.00%	100.00%	100.00%	NEQV
36	100.00%	100.00%	100.00%	100.00%	CALL
37	100.00%	100.00%	100.00%	100.00%	FOR
38	100.00%	100.00%	100.00%	100.00%	ENDFOR
39	100.00%	100.00%	100.00%	100.00%	REPEATUNTIL
40	100.00%	100.00%	100.00%	100.00%	REPEATWHILE
41	100.00%	100.00%	100.00%	100.00%	JUMP
42	100.00%	100.00%	100.00%	100.00%	REPEAT
43	100.00%	100.00%	100.00%	100.00%	RETURN
44	100.00%	100.00%	100.00%	100.00%	GOTO
45	100.00%	100.00%	100.00%	100.00%	FINISH
46	100.00%	100.00%	100.00%	100.00%	IFWHILE
47	100.00%	100.00%	100.00%	100.00%	UNLESSUNTIL
48	100.00%	100.00%	100.00%	100.00%	SWITCH
49	89.04%	89.04%	89.04%	89.04%	IMMEDIATEBYTEARG1
50	88.04%	78.39%	88.04%	78.39%	LOCALPUSHBYTEARG1
51	89.69%	70.31%	89.69%	70.31%	GLOBALPUSHBYTEARG1
52	91.54%	64.36%	91.54%	64.36%	CALLBYTEARG1
53	90.88%	58.49%	90.88%	58.49%	LOCALPOPBYTEARG1
54	95.61%	55.92%	95.61%	55.92%	IFWHILEBYTEARG1
55	96.74%	54.10%	96.74%	54.10%	GLOBALPOPBYTEARG1
56	96.70%	52.31%	96.70%	52.31%	JUMPBYTEARG1

57	97.44%	50.97%	97.27%	50.88%	GLOBALPUSHBYTEARG1CALLBYTEARG1
58	97.20%	49.55%	97.19%	49.46%	IMMEDIATEBYTEARG1ARG1=0
59	97.25%	48.18%	97.25%	48.10%	STATICPUSHBYTEARG1
60	98.03%	47.23%	98.02%	47.14%	STATICPUSHLEFTHALFWORDARG1
61	98.19%	46.38%	98.19%	46.29%	IMMEDIATEARG1=4294967295
62	98.35%	45.61%	98.34%	45.52%	LOCALPUSHBYTEARG1ARG1=0
63	98.50%	44.93%	98.50%	44.84%	GLOBALPUSHHALFWORDARG1
64	98.61%	44.30%	98.60%	44.21%	IMMEDIATEBYTEARG1ARG1=1
65	98.68%	43.72%	98.68%	43.63%	PLUSPUSH
66	98.70%	43.15%	98.70%	43.06%	UNLESSUNTILBYTEARG1
67	98.96%	42.70%	98.74%	42.52%	LOCALPOPBYTEARG1LOCALPUSHBYTEARG1
68	98.86%	42.21%	98.86%	42.03%	ENDFORARG2=1
69	99.00%	41.79%	99.04%	41.63%	LOCALPUSHBYTEARG1ARG1=1
70	99.05%	41.39%	99.05%	41.23%	FORBYTEARG1
71	99.04%	41.00%	99.04%	40.83%	FORBYTEARG1BYTEARG2
72	99.10%	40.63%	99.10%	40.47%	EQIFWHILEBYTEARG1
73	99.11%	40.26%	99.10%	40.10%	ENDFORARG2=1BYTEARG1
74	99.10%	39.90%	99.10%	39.74%	FORBYTEARG1BYTEARG2BYTEARG3
75	99.14%	39.56%	99.15%	39.41%	STATICPUSHBYTEARG1CALLBYTEARG1
76	99.16%	39.23%	99.16%	39.08%	ENDFORARG2=1BYTEARG1BYTEARG3
77	99.17%	38.90%	99.17%	38.75%	IMMEDIATEHALFWORDARG1
78	99.15%	38.57%	99.18%	38.43%	LOCALPUSHBYTEARG1ARG1=2
79	99.18%	38.25%	99.18%	38.12%	STATICPUSHLEFTHALFWORDARG1BYTEARG1
80	99.18%	37.94%	99.17%	37.80%	JUMPHALFWORDARG1
81	99.27%	37.66%	99.27%	37.53%	REPEATBYTEARG1
82	99.34%	37.41%	99.33%	37.28%	LOCALPUSHLEFTBYTEARG1
83	99.35%	37.17%	99.35%	37.03%	PLUSPOP
84	99.16%	36.86%	99.38%	36.81%	LOCALPOPBYTEARG1ARG1=0
85	99.39%	36.63%	99.39%	36.58%	IMMEDIATEBYTEARG1ARG1=2
86	99.40%	36.41%	99.42%	36.37%	LOCALPUSHBYTEARG1ARG1=3
87	99.37%	36.18%	99.47%	36.18%	LOCALPOPBYTEARG1ARG1=1
88	99.25%	35.91%	99.50%	36.00%	GLOBALPUSHHALFWORDARG1CALLBYTEARG1
89	99.59%	35.76%	99.52%	35.82%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=0
90	99.52%	35.59%	99.54%	35.66%	LOCALPUSHBYTEARG1ARG1=4
91	99.51%	35.42%	99.54%	35.50%	LOCALPOPBYTEARG1ARG1=2
92	99.65%	35.29%	99.58%	35.34%	GLOBALPUSHBYTEARG1IMMEDIATEBYTEARG1ARG1=0
93	99.59%	35.15%	99.59%	35.20%	IMMEDIATEBYTEARG1ARG1=3
94	99.61%	35.01%	99.61%	35.06%	SWITCHBYTEARG2
95	99.61%	34.87%	99.62%	34.93%	LOCALPUSHBYTEARG1ARG1=5
96	99.67%	34.76%	99.62%	34.79%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=1
97	99.65%	34.64%	99.62%	34.66%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=2

98	99.56%	34.48%	99.63%	34.53%	GLOBALPUSHBYTEARG1IMMEDIATEBYTEARG1
99	99.66%	34.37%	99.63%	34.40%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=3
100	99.66%	34.25%	99.64%	34.28%	LOCALPOPBYTEARG1ARG1=3
101	99.73%	34.16%	99.67%	34.17%	LOCALPUSHBYTEARG1IMMEDIATEBYTEARG1ARG1=0
102	99.69%	34.05%	99.68%	34.06%	EQUNLESSUNTILBYTEARG1
103	99.71%	33.95%	99.70%	33.96%	LOCALPOPBYTEARG1ARG1=4
104	99.74%	33.86%	99.71%	33.86%	IMMEDIATEBYTEARG1ARG1=10
105	99.67%	33.75%	99.71%	33.76%	LOCALPUSHBYTEARG1ARG1=6
106	99.76%	33.67%	99.71%	33.66%	LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG2=1
107	99.58%	33.53%	99.72%	33.57%	IMMEDIATEBYTEARG1ARG1=0EQIFWHILEBYTEARG1
108	99.73%	33.44%	99.73%	33.47%	SWITCHBYTEARG2HALFWORDARG1
109	99.73%	33.35%	99.73%	33.38%	IMMEDIATEBYTEARG1ARG1=4
110	99.77%	33.27%	99.73%	33.29%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=4
111	99.73%	33.18%	99.73%	33.21%	EQLOGOR
112	99.74%	33.09%	99.74%	33.12%	NEIFWHILEBYTEARG1
113	99.74%	33.01%	99.74%	33.03%	LOGANDIFWHILEBYTEARG1
114	99.75%	32.92%	99.74%	32.95%	STATICPUSHHALFWORDARG1
115	99.75%	32.84%	99.75%	32.86%	REPEATHALFWORDARG1
116	99.72%	32.75%	99.75%	32.78%	GLOBALPUSHBYTEARG1GLOBALPUSHBYTEARG1
117	99.77%	32.67%	99.75%	32.70%	LOCALPUSHBYTEARG1ARG1=0IMMEDIATEBYTEARG1ARG1=0
118	99.80%	32.61%	99.76%	32.62%	LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG2=2
119	99.83%	32.55%	99.76%	32.55%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=5
120	99.78%	32.48%	99.77%	32.47%	LOCALPOPBYTEARG1ARG1=5
121	99.87%	32.44%	99.77%	32.40%	IMMEDIATEBYTEARG1EQIFWHILEBYTEARG1
122	99.55%	32.30%	99.78%	32.33%	IMMEDIATEBYTEARG1ARG1=0GLOBALPOPBYTEARG1
123	100.00%	32.30%	99.78%	32.25%	JUMPBYTEARG1RETURN
124	99.79%	32.23%	99.79%	32.19%	GLOBALPOPHALFWORDARG1
125	99.96%	32.21%	99.79%	32.12%	IMMEDIATEBYTEARG1ARG1=0PLUSPUSH
126	99.83%	32.16%	99.79%	32.05%	LOCALPUSHBYTEARG1IMMEDIATEBYTEARG1
127	99.83%	32.10%	99.80%	31.99%	STATICPUSHBYTEARG1GOTO
128	99.65%	31.99%	99.80%	31.92%	GLOBALPUSHBYTEARG1PLUSPUSH
129	99.70%	31.89%	99.81%	31.86%	GLOBALPUSHBYTEARG1IMMEDIATEBYTEARG1ARG1=1
130	99.81%	31.83%	99.81%	31.80%	STATICPOPBYTEARG1
131	100.00%	31.83%	99.81%	31.74%	LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=0
132	99.79%	31.77%	99.82%	31.69%	LOCALPUSHBYTEARG1ARG1=0IMMEDIATEBYTEARG1
133	99.83%	31.71%	99.82%	31.63%	LOCALPOPBYTEARG1ARG1=6
134	99.83%	31.66%	99.82%	31.57%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=6
135	99.83%	31.60%	99.83%	31.52%	IFWHILEHALFWORDARG1
136	99.81%	31.54%	99.83%	31.47%	LOCALPUSHBYTEARG1ARG1=8
137	99.81%	31.48%	99.83%	31.41%	LOCALPUSHBYTEARG1ARG1=7
138	99.91%	31.46%	99.83%	31.36%	GLOBALPOPBYTEARG1IMMEDIATEBYTEARG1ARG1=0

139	99.99%	31.45%	99.84%	31.31%	LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=1
140	99.84%	31.40%	99.84%	31.26%	GRIFWHILEBYTEARG1
141	99.82%	31.35%	99.84%	31.21%	GLOBALPUSHBYTEARG1GLOBALPUSHBYTEARG1CALLBYTEARG1
142	99.99%	31.35%	99.85%	31.16%	IMMEDIATEBYTEARG1PLUSPUSH
143	100.00%	31.35%	99.85%	31.12%	JUMPBYTEARG1IMMEDIATEBYTEARG1ARG1=0
144	99.89%	31.31%	99.85%	31.07%	IMMEDIATEBYTEARG1GLOBALPOPBYTEARG1
145	99.65%	31.20%	99.85%	31.02%	LOCALPUSHBYTEARG1ARG1=0PLUSPUSH
146	99.94%	31.18%	99.85%	30.98%	IMMEDIATEBYTEARG1ARG1=1EQIFWHILEBYTEARG1
147	99.84%	31.13%	99.85%	30.93%	PLUSGLOBALPOPBYTEARG1
148	99.80%	31.07%	99.85%	30.89%	LOCALPUSHBYTEARG1PLUSPUSH
149	99.85%	31.03%	99.85%	30.84%	GLOBALPUSHLEFTBYTEARG1
150	99.99%	31.02%	99.86%	30.80%	LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=2
151	99.86%	30.98%	99.87%	30.76%	STATICPUSHBYTEARG1CALLBYTEARG1ARG2=0
152	99.99%	30.97%	99.87%	30.72%	STATICPUSHLEFTHALFWORDARG1BYTEARG1GLOBALPUSHBYTEARG1
153	100.00%	30.97%	99.87%	30.68%	LOCALPOPBYTEARG1FORBYTEARG1BYTEARG2BYTEARG3
154	99.77%	30.90%	99.87%	30.64%	LOCALPOPBYTEARG1IMMEDIATEBYTEARG1ARG1=0
155	99.85%	30.86%	99.87%	30.60%	IMMEDIATEBYTEARG1ARG1=5
156	99.95%	30.84%	99.87%	30.56%	GLOBALPUSHBYTEARG1CALLBYTEARG1RETURN
157	99.87%	30.80%	99.88%	30.52%	IMMEDIATEBYTEARG1ARG1=8
158	99.97%	30.79%	99.88%	30.49%	LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=0
159	99.86%	30.75%	99.88%	30.45%	IMMEDIATEBYTEARG1ARG1=7
160	99.93%	30.73%	99.88%	30.41%	LOCALPUSHBYTEARG1ARG1=1IMMEDIATEBYTEARG1ARG1=0
161	99.83%	30.68%	99.88%	30.38%	GLOBALPUSHBYTEARG1GLOBALPOPBYTEARG1
162	99.90%	30.65%	99.88%	30.34%	IMMEDIATEBYTEARG1ARG1=1PLUSPUSH
163	99.92%	30.62%	99.88%	30.31%	GLOBALPUSHBYTEARG1CALLBYTEARG1JUMPBYTEARG1
164	99.89%	30.59%	99.88%	30.27%	LOCALPUSHBYTEARG1ARG1=9
165	99.87%	30.55%	99.89%	30.24%	LOCALPUSHBYTEARG1ARG1=11
166	99.95%	30.54%	99.89%	30.20%	IMMEDIATEBYTEARG1ARG1=1GLOBALPOPBYTEARG1
167	99.87%	30.50%	99.89%	30.17%	IMMEDIATEBYTEARG1ARG1=6
168	99.97%	30.49%	99.89%	30.14%	GLOBALPUSHBYTEARG1PLUS
169	99.89%	30.45%	99.89%	30.10%	ENDFORBYTEARG1
170	99.89%	30.42%	99.89%	30.07%	LSIFWHILEBYTEARG1
171	99.89%	30.39%	99.89%	30.04%	IMMEDIATEBYTEARG1ARG1=0EQ
172	99.88%	30.35%	99.89%	30.01%	STATICPUSHBYTEARG1CALLBYTEARG1ARG1=0
173	99.88%	30.32%	99.89%	29.97%	IMMEDIATEBYTEARG1ARG1=32
174	99.86%	30.27%	99.89%	29.94%	GLOBALPUSHBYTEARG1IFWHILEBYTEARG1
175	99.89%	30.24%	99.89%	29.91%	UNLESSUNTILHALFWORDARG1
176	99.91%	30.21%	99.89%	29.88%	LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=3
177	99.89%	30.18%	99.89%	29.84%	REPEATUNTILBYTEARG1
178	99.86%	30.14%	99.89%	29.81%	IMMEDIATEBYTEARG1ARG1=20

179	99.67%	30.04%	99.89%	29.78%	STATICPUSHLEFTHALFWORDARG1BYTEARG1GLOBALPUSHBYTEARG1CALLBYTEARG
180	99.92%	30.01%	99.90%	29.75%	STATICPUSHHALFWORDARG1CALLBYTEARG1
181	99.89%	29.98%	99.90%	29.72%	EQLOGORIFWHILEBYTEARG1
182	99.99%	29.98%	99.90%	29.69%	IMMEDIATEBYTEARG1ARG1=0PLUS
183	99.90%	29.95%	99.90%	29.66%	ENDFORBYTEARG1BYTEARG3
184	99.91%	29.92%	99.90%	29.63%	LOCALPOPBYTEARG1ARG1=7
185	99.91%	29.89%	99.90%	29.60%	LOCALPUSHBYTEARG1ARG1=0IMMEDIATEBYTEARG1ARG1=1
186	99.84%	29.85%	99.91%	29.57%	GLOBALPUSHHALFWORDARG1CALLBYTEARG1ARG1=256
187	99.94%	29.83%	99.91%	29.55%	STATICPUSHBYTEARG1CALLBYTEARG1ARG2=2
188	99.92%	29.80%	99.91%	29.52%	STATICPUSHBYTEARG1CALLBYTEARG1ARG2=1
189	99.90%	29.77%	99.91%	29.49%	STATICPUSHBYTEARG1CALLBYTEARG1ARG2=3
190	100.00%	29.77%	99.91%	29.46%	IMMEDIATEBYTEARG1ARG1=0MINUS
191	99.93%	29.75%	99.91%	29.44%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=7
192	99.90%	29.72%	99.91%	29.41%	LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=4
193	99.92%	29.70%	99.91%	29.39%	LELOGAND
194	99.91%	29.67%	99.91%	29.36%	SWITCHBYTEARG2HALFWORDARG1BYTEARG1
195	99.87%	29.64%	99.91%	29.33%	LOCALPUSHBYTEARG1IMMEDIATEBYTEARG1ARG1=1
196	99.91%	29.61%	99.91%	29.31%	STATICPUSHLEFTHALFWORDARG1BYTEARG1JUMPBYTEARG1
197	99.98%	29.60%	99.92%	29.28%	GLOBALPUSHHALFWORDARG1ARG1=256
198	99.98%	29.60%	99.92%	29.26%	LOCALPUSHBYTEARG1ARG1=0PLUS
199	99.88%	29.56%	99.92%	29.23%	IMMEDIATEBYTEARG1ARG1=48
200	99.94%	29.54%	99.92%	29.21%	GLOBALPUSHBYTEARG1PLUSPOP
201	99.86%	29.50%	99.92%	29.18%	LOCALPUSHBYTEARG1ARG1=1PLUSPUSH
202	99.93%	29.48%	99.92%	29.16%	LOCALPUSHBYTEARG1ARG1=2IMMEDIATEBYTEARG1ARG1=0
203	99.92%	29.45%	99.92%	29.14%	FORBYTEARG1BYTEARG2HALFWORDARG3
204	99.90%	29.42%	99.92%	29.11%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=9
205	99.92%	29.40%	99.92%	29.09%	IMMEDIATEBYTEARG1ARG1=9
206	99.92%	29.38%	99.92%	29.07%	LOCALPUSHBYTEARG1PLUS
207	99.92%	29.35%	99.92%	29.04%	LOCALPOPBYTEARG1ARG1=8
208	99.96%	29.34%	99.92%	29.02%	GLOBALPUSHHALFWORDARG1CALLBYTEARG1ARG1=259
209	99.83%	29.29%	99.92%	29.00%	IMMEDIATEBYTEARG1ARG1=0JUMPBYTEARG1
210	99.96%	29.28%	99.93%	28.98%	LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG2=3
211	99.95%	29.27%	99.93%	28.96%	GLOBALPOPBYTEARG1RETURN
212	99.94%	29.25%	99.93%	28.94%	LOCALPUSHBYTEARG1ARG1=13
213	99.93%	29.23%	99.93%	28.91%	ENDFORARG2=1BYTEARG1HALFWORDARG3
214	99.93%	29.21%	99.93%	28.89%	LOGORIFWHILEBYTEARG1
215	99.97%	29.20%	99.93%	28.87%	IMMEDIATEBYTEARG1ARG1=0PLUSPOP
216	99.93%	29.18%	99.93%	28.85%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=10
217	99.95%	29.16%	99.93%	28.83%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=8

218	99.90%	29.13%	99.93%	28.81%	GLOBALPUSHBYTEARG1IMMEDIATEBYTEARG1ARG1=2
219	99.92%	29.11%	99.93%	28.79%	REPEATWHILEBYTEARG1
220	99.94%	29.09%	99.93%	28.77%	LOCALPUSHBYTEARG1ARG1=10
221	100.00%	29.09%	99.93%	28.75%	JUMPBYTEARG1STATICPUSHLEFTHALFWORDARG1BYTEARG1
222	100.00%	29.09%	99.93%	28.73%	LOCALPOPBYTEARG1ARG1=0FORBYTEARG1BYTEARG2BYTEARG3
223	99.90%	29.06%	99.93%	28.71%	GLOBALPUSHHALFWORDARG1ARG1=259
224	99.93%	29.04%	99.93%	28.70%	ENDFORBYTEARG1BYTEARG3ARG2=4294967295
225	99.94%	29.03%	99.93%	28.68%	LOCALPUSHBYTEARG1ARG1=2PLUSPUSH
226	99.90%	29.00%	99.93%	28.66%	IMMEDIATEBYTEARG1ARG1=0EQUNLESSUNTILBYTEARG1
227	99.97%	28.99%	99.93%	28.64%	IMMEDIATEBYTEARG1ARG1=1EQ
228	100.00%	28.99%	99.93%	28.62%	GLOBALPUSHBYTEARG1ARG1=74
229	99.93%	28.97%	99.93%	28.60%	GEIFWHILEBYTEARG1
230	100.00%	28.97%	99.93%	28.58%	JUMPHALFWORDARG1RETURN
231	99.92%	28.95%	99.93%	28.56%	STATICPUSHBYTEARG1CALLBYTEARG1ARG2=4
232	99.86%	28.91%	99.93%	28.54%	GLOBALPUSHHALFWORDARG1CALLBYTEARG1ARG2=0
233	99.91%	28.88%	99.94%	28.53%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=12
234	99.96%	28.87%	99.94%	28.51%	LOCALPUSHBYTEARG1ARG1=12
235	99.94%	28.85%	99.94%	28.49%	LOCALPOPBYTEARG1ARG1=9
236	99.95%	28.84%	99.94%	28.47%	GLOBALPOPBYTEARG1ARG1=10
237	99.96%	28.83%	99.94%	28.45%	GLOBALPUSHBYTEARG1IMMEDIATEARG1=4294967295
238	100.00%	28.83%	99.94%	28.44%	JUMPBYTEARG1LOCALPUSHBYTEARG1ARG1=0
239	99.93%	28.81%	99.94%	28.42%	POPBYTEENDFORARG2=1BYTEARG1BYTEARG3
240	99.96%	28.79%	99.94%	28.40%	LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG2=4
241	99.70%	28.71%	99.94%	28.38%	IMMEDIATEBYTEARG1ARG1=1PLUS
242	99.93%	28.69%	99.94%	28.37%	LOCALPUSHBYTEARG1ARG1=0PLUSPOP
243	99.96%	28.68%	99.94%	28.35%	NELOGAND
244	100.00%	28.68%	99.94%	28.33%	GLOBALPUSHBYTEARG1ARG1=73
245	99.95%	28.66%	99.94%	28.32%	IMMEDIATEBYTEARG1EQ
246	99.95%	28.65%	99.94%	28.30%	STATICPUSHBYTEARG1CALLBYTEARG1ARG2=5
247	99.93%	28.63%	99.94%	28.28%	LELOGANDIFWHILEBYTEARG1
248	99.98%	28.62%	99.94%	28.27%	IMMEDIATEBYTEARG1ARG1=2EQIFWHILEBYTEARG1
249	100.00%	28.62%	99.94%	28.25%	LOCALPOPBYTEARG1ARG1=1FORBYTEARG1BYTEARG2BYTEARG3
250	99.94%	28.61%	99.94%	28.23%	LOCALPUSHLEFTBYTEARG1ARG1=1
251	99.93%	28.59%	99.94%	28.22%	MINUSGLOBALPOPBYTEARG1
252	100.00%	28.59%	99.94%	28.20%	JUMPBYTEARG1LOCALPUSHBYTEARG1
253	99.87%	28.55%	99.94%	28.19%	IMMEDIATEBYTEARG1ARG1=0NEIFWHILEBYTEARG1
254	100.00%	28.55%	99.94%	28.17%	LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG2=1ARG1=0
255	99.95%	28.53%	99.94%	28.16%	IMMEDIATEARG1=4294967295EQIFWHILEBYTEARG1

This is the distribution of instructions in the optimised code.

Freq.	*Instruction*
3.6201%	GLOBALPUSHBYTEARG1CALLBYTEARG1
3.4453%	JUMPBYTEARG1
3.2895%	GLOBALPUSHBYTEARG1
2.9595%	LOCALPUSHBYTEARG1
2.8163%	GLOBALPUSHHALFWORDARG1CALLBYTEARG1
2.6268%	IMMEDIATEBYTEARG1
2.2997%	LOCALPOPBYTEARG1LOCALPUSHBYTEARG1
2.1228%	LOCALPOPBYTEARG1
1.8694%	IMMEDIATEBYTEARG1ARG1=0
1.7163%	IMMEDIATEHALFWORDARG1
1.6876%	FORBYTEARG1BYTEARG2BYTEARG3
1.6721%	STATICPUSHLEFTHALFWORDARG1
1.6468%	JUMPHALFWORDARG1
1.6440%	GLOBALPOPBYTEARG1
1.6195%	STATICPUSHBYTEARG1CALLBYTEARG1
1.5984%	GLOBALPUSHBYTEARG1IMMEDIATEBYTEARG1
1.5977%	LOCALPUSHBYTEARG1ARG1=0
1.4068%	STATICPUSHLEFTHALFWORDARG1BYTEARG1GLOBALPUSHBYTEARG1CALLBYTEARG1
1.2987%	STATICPUSHLEFTHALFWORDARG1BYTEARG1
1.2579%	LOCALPUSHBYTEARG1ARG1=1
1.0993%	LOCALPOPBYTEARG1ARG1=0
1.0853%	IMMEDIATEBYTEARG1ARG1=1
1.0761%	RETURN
1.0368%	LOCALPUSHBYTEARG1ARG1=2
1.0347%	EQIFWHILEBYTEARG1
1.0277%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=0
1.0122%	IMMEDIATEBYTEARG1ARG1=0GLOBALPOPBYTEARG1
0.9856%	IMMEDIATEBYTEARG1ARG1=0EQIFWHILEBYTEARG1
0.9519%	GLOBALPUSHBYTEARG1GLOBALPUSHBYTEARG1
0.9435%	ENDFORARG2=1BYTEARG1BYTEARG3
0.8789%	GLOBALPUSHBYTEARG1IMMEDIATEBYTEARG1ARG1=0
0.8634%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=2
0.8283%	IFWHILEBYTEARG1
0.8241%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=3
0.8017%	LOCALPOPBYTEARG1ARG1=1
0.7960%	GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=1

```
0.7918% GLOBALPUSHBYTEARG1PLUSPUSH
0.7750% GLOBALPUSHBYTEARG1GLOBALPUSHBYTEARG1CALLBYTEARG1
0.7736% LOCALPUSHBYTEARG1ARG1=3
0.7272% PLUS
0.6809% GLOBALPUSHBYTEARG1IMMEDIATEBYTEARG1ARG1=1
0.6760% PLUSPOP
0.6739% PLUSPUSH
0.6676% IMMEDIATEBYTEARG1ARG1=2
0.6570% IMMEDIATEARG1=4294967295
0.6542% LOCALPUSHBYTEARG1IMMEDIATEBYTEARG1ARG1=0
0.6472% REPEATBYTEARG1
0.6107% LOCALPOPBYTEARG1ARG1=2
0.5988% LOCALPUSHBYTEARG1ARG1=4
0.5833% LOCALPUSHBYTEARG1IMMEDIATEBYTEARG1
0.5742% UNLESSUNTILBYTEARG1
0.5602% LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG2=1
0.5475% GLOBALPUSHHALFWORDARG1
0.5461% EQUNLESSUNTILBYTEARG1
0.5433% MINUS
0.5377% GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=4
0.5370% GLOBALPUSHBYTEARG1GLOBALPOPBYTEARG1
0.5096% IMMEDIATEBYTEARG1ARG1=3
0.4900% LOCALPOPBYTEARG1IMMEDIATEBYTEARG1ARG1=0
0.4816% LOCALPUSHBYTEARG1ARG1=5
0.4731% LOCALPUSHLEFTBYTEARG1
0.4703% LOGANDIFWHILEBYTEARG1
0.4647% LOCALPUSHBYTEARG1ARG1=0IMMEDIATEBYTEARG1
0.4570% IMMEDIATEBYTEARG1EQIFWHILEBYTEARG1
0.4493% LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG2=2
0.4401% PUSHBYTE
0.4380% REPEATHALFWORDARG1
0.4380% LOCALPUSHBYTEARG1PLUSPUSH
0.4380% GLOBALPUSHHALFWORDARG1CALLBYTEARG1ARG2=0
0.4317% GLOBALPUSHBYTEARG1IFWHILEBYTEARG1
0.4142% EQ
0.4100% LOCALPOPBYTEARG1ARG1=3
0.3959% GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=5
0.3952% LOCALPUSHBYTEARG1ARG1=6
0.3945% STATICPUSHBYTEARG1
0.3847% LOCALPUSHBYTEARG1ARG1=0PLUSPUSH
0.3720% STATICPUSHBYTEARG1GOTO
```

0.3699% LOGAND
0.3692% GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=6
0.3685% IMMEDIATEBYTEARG1GLOBALPOPBYTEARG1
0.3643% GLOBALPOPHALFWORDARG1
0.3580% IMMEDIATEBYTEARG1ARG1=0JUMPBYTEARG1
0.3566% NEIFWHILEBYTEARG1
0.3482% GLOBALPUSHBYTEARG1CALLBYTEARG1JUMPBYTEARG1
0.3447% LOCALPOPBYTEARG1ARG1=4
0.3426% GRIFWHILEBYTEARG1
0.3412% PLUSGLOBALPOPBYTEARG1
0.3229% STATICPUSHHALFWORDARG1CALLBYTEARG1
0.3138% IMMEDIATEBYTEARG1ARG1=4
0.3131% STATICPUSHBYTEARG1CALLBYTEARG1ARG2=0
0.3089% IMMEDIATEBYTEARG1ARG1=10
0.2990% IMMEDIATEBYTEARG1ARG1=1PLUS
0.2906% STATICPUSHLEFTHALFWORDARG1BYTEARG1JUMPBYTEARG1
0.2885% IFWHILEHALFWORDARG1
0.2780% SWITCHBYTEARG2HALFWORDARG1
0.2759% SWITCHBYTEARG2HALFWORDARG1BYTEARG1
0.2660% PUSH
0.2639% LOCALPUSHBYTEARG1IMMEDIATEBYTEARG1ARG1=1
0.2597% IMMEDIATEBYTEARG1ARG1=0NEIFWHILEBYTEARG1
0.2590% LOCALPUSHBYTEARG1ARG1=0IMMEDIATEBYTEARG1ARG1=0
0.2555% STATICPUSHBYTEARG1CALLBYTEARG1ARG1=0
0.2492% IMMEDIATE
0.2464% LOCALPOPBYTEARG1ARG1=5
0.2331% LSIFWHILEBYTEARG1
0.2232% EQLOGORIFWHILEBYTEARG1
0.2190% POPBYTEENDFORARG2=1BYTEARG1BYTEARG3
0.2120% LOCALPUSHBYTEARG1ARG1=8
0.2106% FORBYTEARG1BYTEARG2HALFWORDARG3
0.2092% STATICPUSHBYTEARG1CALLBYTEARG1ARG2=3
0.2092% GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=9
0.2064% LE
0.2050% LOCALPUSHBYTEARG1ARG1=7
0.2050% IMMEDIATEBYTEARG1ARG1=0EQUNLESSUNTILBYTEARG1
0.2043% NE
0.2022% GLOBALPOPBYTEARG1IMMEDIATEBYTEARG1ARG1=0
0.2022% LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=4
0.2008% GLOBALPUSHBYTEARG1IMMEDIATEBYTEARG1ARG1=2
0.2001% STATICPUSHHALFWORDARG1

0.1994% EQLOGOR
0.1994% LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=3
0.1895% LOCALPOPBYTEARG1ARG1=6
0.1755% GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=12
0.1741% STATICPUSHBYTEARG1CALLBYTEARG1ARG2=1
0.1699% GLOBALPUSHHALFWORDARG1CALLBYTEARG1ARG1=256
0.1685% UNLESSUNTILHALFWORDARG1
0.1664% IMMEDIATEBYTEARG1ARG1=5
0.1615% LOCALPUSHBYTEARG1PLUS
0.1558% GLOBALPUSHBYTEARG1CALLBYTEARG1RETURN
0.1558% STATICPUSHBYTEARG1CALLBYTEARG1ARG2=4
0.1544% GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=7
0.1523% IMMEDIATEBYTEARG1ARG1=7
0.1516% IMMEDIATEBYTEARG1ARG1=20
0.1502% MINUSGLOBALPOPBYTEARG1
0.1488% ENDFORARG2=1BYTEARG1HALFWORDARG3
0.1481% LOCALPUSHBYTEARG1ARG1=1PLUSPUSH
0.1432% STATICPOPBYTEARG1
0.1432% LOCALPUSHBYTEARG1ARG1=11
0.1390% LOGOR
0.1390% IMMEDIATEBYTEARG1ARG1=8
0.1390% LOGORIFWHILEBYTEARG1
0.1362% IMMEDIATEBYTEARG1ARG1=6
0.1348% GEIFWHILEBYTEARG1
0.1334% GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=10
0.1334% LELOGANDIFWHILEBYTEARG1
0.1327% ENDFORBYTEARG1BYTEARG3
0.1320% GLOBALPUSHBYTEARG1PLUSPOP
0.1292% IMMEDIATEBYTEARG1ARG1=48
0.1264% MULT
0.1264% IMMEDIATEBYTEARG1ARG1=1EQIFWHILEBYTEARG1
0.1257% NOT
0.1250% POPBYTE
0.1250% IMMEDIATEBYTEARG1ARG1=32
0.1179% STATICPUSHBYTEARG1CALLBYTEARG1ARG2=2
0.1172% IMMEDIATEBYTEARG1ARG1=0EQ
0.1144% LOCALPUSHBYTEARG1ARG1=9
0.1095% STATICPUSHBYTEARG1CALLBYTEARG1ARG2=5
0.1095% IMMEDIATEARG1=4294967295EQIFWHILEBYTEARG1
0.1088% DIV
0.1088% IMMEDIATEBYTEARG1ARG1=1PLUSPUSH

0.1081% GLOBALPUSHBYTEARG1CALLBYTEARG1ARG2=8
0.1053% RSHIFT
0.1053% GLOBALPUSHLEFTBYTEARG1
0.1039% GLOBALPOPBYTEARG1RETURN
0.1039% IMMEDIATEBYTEARG1EQ
0.0990% LOCALPOPBYTEARG1ARG1=7
0.0976% POP
0.0969% IMMEDIATEBYTEARG1ARG1=1GLOBALPOPBYTEARG1
0.0906% LOCALPUSHBYTEARG1ARG1=0IMMEDIATEBYTEARG1ARG1=1
0.0884% GR
0.0870% LOCALPOPBYTEARG1ARG1=8
0.0828% LELOGAND
0.0793% IMMEDIATEBYTEARG1ARG1=9
0.0786% REPEATUNTILBYTEARG1
0.0786% LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG2=4
0.0744% LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG2=3
0.0744% GLOBALPUSHBYTEARG1IMMEDIATEARG1=4294967295
0.0730% LOCALPUSHBYTEARG1ARG1=1IMMEDIATEBYTEARG1ARG1=0
0.0730% LOCALPUSHBYTEARG1ARG1=0PLUSPOP
0.0709% LSHIFT
0.0709% LOCALPUSHBYTEARG1ARG1=2IMMEDIATEBYTEARG1ARG1=0
0.0695% QUERY
0.0660% LOCALPUSHBYTEARG1ARG1=13
0.0646% LOCALPUSHBYTEARG1ARG1=10
0.0625% GOTO
0.0618% LS
0.0618% GLOBALPUSHBYTEARG1PLUS
0.0590% LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=0
0.0576% LOCALPOPBYTEARG1ARG1=9
0.0562% LOCALPUSHBYTEARG1ARG1=2PLUSPUSH
0.0555% LOCALPUSHLEFTBYTEARG1ARG1=1
0.0541% GLOBALPOPBYTEARG1ARG1=10
0.0534% REPEATWHILEBYTEARG1
0.0526% GLOBALPUSHHALFWORDARG1ARG1=259
0.0526% ENDFORBYTEARG1BYTEARG3ARG2=4294967295
0.0505% REM
0.0491% IMMEDIATEBYTEARG1ARG1=0PLUSPUSH
0.0484% NEG
0.0463% GE
0.0428% LOCALPUSHBYTEARG1ARG1=12
0.0407% GLOBALPUSHHALFWORDARG1CALLBYTEARG1ARG1=259

```
0.0407% NELOGAND
0.0365% CALLBYTEARG1
0.0337% STATICPUSHLEFTHALFWORDARG1BYTEARG1GLOBALPUSHBYTEARG1
0.0337% IMMEDIATEBYTEARG1ARG1=0PLUSPOP
0.0323% IMMEDIATEBYTEARG1ARG1=2EQIFWHILEBYTEARG1
0.0309% LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=2
0.0281% ENDFORBYTEARG1
0.0267% IMMEDIATEBYTEARG1ARG1=1EQ
0.0239% IMMEDIATEBYTEARG1PLUSPUSH
0.0225% LOCALPUSHBYTEARG1ARG1=0PLUS
0.0211% REPEATUNTIL
0.0204% ABS
0.0126% LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=1
0.0105% GLOBALPUSHLEFT
0.0105% STATICPOP
0.0098% LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG1=0
0.0098% IMMEDIATEBYTEARG1ARG1=0PLUS
0.0084% FINISH
0.0084% GLOBALPUSHHALFWORDARG1ARG1=256
0.0049% NEQV
0.0035% REPEATWHILE
0.0021% LOCALPOPBYTEARG1LOCALPUSHBYTEARG1ARG2=1ARG1=0
0.0014% EQV
0.0014% JUMPBYTEARG1LOCALPUSHBYTEARG1ARG1=0
0.0007% IMMEDIATEBYTEARG1ARG1=0MINUS
0.0000% UNKNOWN
0.0000% GLOBALPUSH
0.0000% STATICPUSH
0.0000% LOCALPUSH
0.0000% STATICPUSHLEFT
0.0000% LOCALPUSHLEFT
0.0000% GLOBALPOP
0.0000% LOCALPOP
0.0000% CALL
0.0000% FOR
0.0000% ENDFOR
0.0000% JUMP
0.0000% REPEAT
0.0000% IFWHILE
0.0000% UNLESSUNTIL
0.0000% SWITCH
```

```
0.0000% ENDFORARG2=1
0.0000% FORBYTEARG1
0.0000% FORBYTEARG1BYTEARG2
0.0000% ENDFORARG2=1BYTEARG1
0.0000% SWITCHBYTEARG2
0.0000% JUMPBYTEARG1RETURN
0.0000% JUMPBYTEARG1IMMEDIATEBYTEARG1ARG1=0
```