ISGEN: A Byte Stream Instruction Set Generator

F D L Arci and J P Bennett
School of Mathematical Sciences
University of Bath
BATH, BA2 7AY

August 6, 1992

Abstract

Various methodologies have been devised for the design of byte stream instruction sets (Tan78, SS82). The second author has proposed an approach that is largely automatic(Ben88). A set of instructions is derived that is optimal according to some criterion, such as the size of compiled code. The choice of instructions is driven by statistical analysis of a large amount of high level language code intended for the instruction set under design. We describe a computer program which will produce such an instruction set. The system has been successfully used to produce bytestream instruction sets to support BCPL (RWS80), Poly(Mat85) and EuLisp (PN+90). We present quantitative results showing the success of these designs. Byte stream instruction sets are now largely restricted to interpretive intermediate codes, with the majority of instruction sets being RISC, or derived designs. We outline current work to produce ISGEN-GA which will generalise the methodology, so that RISC type instruction sets can be produced automatically.

1 Background

An instruction set is the only interface the compiler writer has to control the actions of a computer. We call the difference between the possibilities of low level hardware and high level language constructs the *semantic gap*. This is precisely what an instruction set attempts to bridge.

Instruction set design is concerned with providing efficient ways to bridge this gap and various strategies are viable. Among these we distinguish two main ones: the first (CISC's: Complex Instruction Set Computers) provides big complex instructions to closely reflect the essential operators of a high level language, the second (RISC's: Reduced Instruction Set Computers (PD80)) forms its instruction sets with a small number of very simple instructions and it is up to the compiler writer to choose the best sequence of opcodes that matches semantically a particular high level language construct.

Early designs were very much influenced by what the designer felt ought to be in a language oriented machine. It was not until the 1970's that people started to analyze statistically the use of existing instruction sets in order to provide data with which to design future ones. Various formal design procedures were hence developed. A common feature was the realisation of the need to start with a small core instruction set containing only general purpose instructions and to augment it successively with the necessary specialised opcodes (Tan78, SS82).

Automated techniques that, given the necessary statistics, are able to augment a generic byte stream instruction set[1] have since been developed (Ben88). Although examples in this paper all refer to byte stream sets the methodology presented is general and work is being carried out to apply it to other styles of instruction sets.

2 Designing an instruction set

We use a development of the methodology of Sweet and Sandman (SS82). We give rules to choose an initial instruction set to support a particular language and environment. We then provide automatic techniques to extend that instruction set and thus improve it.

To achieve this we need a number of items:

- We need a clear model of the high level language being used.
- We need a clear model of the target architecture style.
- We need a typical body of code from the environment in which the language is being used, to provide statistical information.
- We need to quantify our criterion for instruction set quality.

A good model of the high level language comes from the designer's experience, but can be helped by statistical analysis of program usage. Most languages boil down to operations to access data from various areas (stack, global area, constants etc.); operations to manipulate that data (arithmetic, logical, relational etc.) and operations to manipulate the flow of control through the program (for loops, goto statements, procedure calls etc.).

The model of the target architecture will govern what instructions are realistic. This may be a philosophical choice (e.g. a decision to build a load/store RISC architecture) or it may be constrained by circumstances (if you have a microcoded architecture which supports byte stream instruction sets, you really have to use a byte stream instruction set).

It is important that we consider a language being used, *within its target environment*. The best instruction set to support C will be different if the language is being used to run business software or if it is being used to run numerical simulations.

Finally if we are to choose new instructions automatically we must have a way of measuring how good an instruction set is. We could choose *entropy* (Abr63): the average number of bits per symbol effectively used in an encoding. Entropy is a good measure as it can be compared directly with the number of bits per symbol actually used, but size of a sample body of compiled code is by far the commonest. A good instruction set is then one which yields small compiled programs. The benefit of adding a new instruction can be quantified by the reduction in size of a sample body of

[1]A byte stream instruction set is one where instructions consist of a single byte opcode specifying the required operation, possibly followed by a number of argument bytes. The number of instructions in such an architecture is limited by the size of a byte, which today almost invariably means there may be up to 256 opcodes. In case more are needed some of the existing ones can be selected to act as 'escape' opcodes and a subsequent byte can then act as a secondary opcode. Throughout the rest of this paper we shall use the term 'opcode' to mean the initial byte specifying the operation to be carried out and 'instruction' to indicate the opcode together with its arguments (BS89)

compiled code. Other criteria can be used, but are often harder to measure. For example program speed involves simulation of bodies of code, and so for each new instruction a simulation must be created (not impossible, but hard).

We will present three examples to illustrate this work, but in each case static code size has been the criterion. The value of this has been justified elsewhere (Bennett, Schoepke), but helps to improve program speed by increasing cache occupancy, reducing working sets and reducing program load times. Each example also is a byte stream instruction set. However it should be made clear that the methodology is equally suited to fixed or variable format instruction sets, provided suitable design rules are given. Indeed ISGEN-GA (discussed in section 5) is working with just such instruction sets.

The methodology

We use a six step methodology.

1. Based on our analysis of the high level language and the target hardware we select a minimal instruction set which can support the language. This is the *canonical instruction set*. We match each high level construct with one or two instructions within the low level architecture. Some constructs may need more than one instruction (typically loops need one at each end). Where instructions have the same semantics they are merged (for example the instruction for an IF statement is the same as the first instruction for a WHILE loop).
2. We write a compiler for the canonical instruction set, and use it to compile a body of code representative of language being used in its target environment.
3. We decide on a quantifiable design criterion. For example compactness of compiled code.
4. We determine a set of rules for creating new instructions to add to the canonical instruction set, which may improve the instruction set according to the design criterion. For example if we are looking for compact compiled code and are working with a byte stream instruction set we could use the rules:
   - Provide a new opcode with a reduce argument range (e.g LOAD-BYTE derived from LOAD-WORD);
   - Provide a new opcode with a specific argument value (e.g. LOAD-CONSTANT-1 derived from LOAD-CONSTANT-WORD);
   - Combine two opcodes (e.g ADD-WORD derived from LOAD-WORD and ADD).

   Note that all these operations lead to instructions which take less space.
5. We collect statistics on the body of code, and using the statistics identify the new instruction, which if created and substituted wherever possible would lead to the biggest improvement according to our design criterion. The instruction is identified by exhaustive searching of all possible instructions.
6. We then peephole optimise this instruction into our body of code.
7. We repeat steps 5–6 until we have sufficient instructions for our instruction set. For a byte-stream this would be when there were 256 instructions in total.

There is scope for refinement of the technology. An optimising compiler would do better than our peepholer with a new instruction for instance. With modern techniques it would be possible to create an instruction definition, rebuild the compiler and recompile the sample code, but the effort is not really worth it. We often create instructions initially, which are rendered less valuable by later instructions. For example we may generate LOAD-CONSTANT-BYTE to load small constants, and then generate LOAD-CONSTANT-ZERO and LOAD-CONSTANT-ONE as two specific cases. However these two cases are almost all the load constants less than one byte, and the existence of load constant byte can no longer be justified.

Ultimately these problems are because we are using a local optimisation, whilst we need a global optimisation.

3 ISGEN-1 and ISGEN-2

ISGEN works by taking a set of statistics on a given instruction set and a set of rules and considering which rule would maximise some specified design criterium (e.g. compactness of compiled code, entropy of the instruction set). To achieve this it exhaustively considers all possibilities of design rule application. The first version of ISGEN, ISGEN-1 adjusts the statistics according to the generated instruction and then repeats the whole process until a prefixed number had been reached.

The deduction of new statistics is unfortunately prone to error and ISGEN-2 now performs peephole substitution of each generated instruction followed by statistics recollection.

ISGEN-2 takes about twenty minutes on a SPARC station to generate 256 instructions from an initial canonical set of 40, using three design rules. The design rules used are the ones outlined in the previous section. Factors that affect performance are:

- the number and complexity of design rules
- the size of the code sample
- the number of instructions needed.

4 Case studies

ISGEN has been successfully used to produce byte stream instruction sets to support BCPL (RWS80), Poly (Mat85) and EuLisp (PN+90). This section quantitatively analyzes its performance in each of these cases.

BCPL

BCPL, as used in the Tripos command environment, is a language closely related to C. It has a very simple structure, but the same basic ideas underlie most imperative programming languages. It is its very simplicity that suggested the possibility of an approach uncluttered by excessive detail.

The target architecture chosen was a High Level Hardware Orion, which is a 32 bit soft microprogrammable mini computer built form standard bit-slice TTL. It supports byte stream instruction sets, with a hardware switch on a byte operand provided in the microengine. The example canonical instruction set is therefore a byte stream instruction set where arguments to all opcodes are 32 bits in length. A set of 48 instructions mapping one to one to high level language concepts was chosen as a canonical set (Appendix A). A compiler from this canonical instruction set was written and 500K of compiled canonical code obtained by compiling the 102 BCPL programs that constitute the Tripos command environment.

Code was optimised for static size. The evaluation of the results was carried out with the aid of a simple peephole optimiser which added the new instructions to the existing code. Code shrunk to 28.53 % of its original size. The synthetic instruction set is non-orthogonal as only instructions that are actually needed are generated.

POLY

ISGEN was used to refine an instruction set for the polymorphic programming language, POLY (Mat85). 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. The existing intermediate code, consisting of 24 instructions was taken as the canonical instruction set. Sample statistics were provided from 214074 bytes of compiled code.

Out of the 232 instructions proposed by ISGEN Matthews accepted only the first 97, responsible for about 85 % of the improvement and incorporated them into his compiler. This new compiler produced 82560 bytes of compiled code, a reduction to 38.57 % of the original code size. These new statistics were then fed back into ISGEN, which proposed a further refinement of the instruction set to achieve a reduction in code size to 29 % of the original size. This result 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.

BEEP: A BytecodE for EulisP

Eulisp is the draft European Lisp System (PN+90). Compilation to a bytecode provides a very convenient and compact way of representing programs so that they can run efficiently in a reasonably small amount of memory.

A canonically compiled code sample of 446051 bytes was used as initial data for the optimization. Code was again optimised for static size and shrunk to 5% of its original size, although the result has to be evaluated in the light of the fact that the canonical instruction set used generates particularly verbose code.

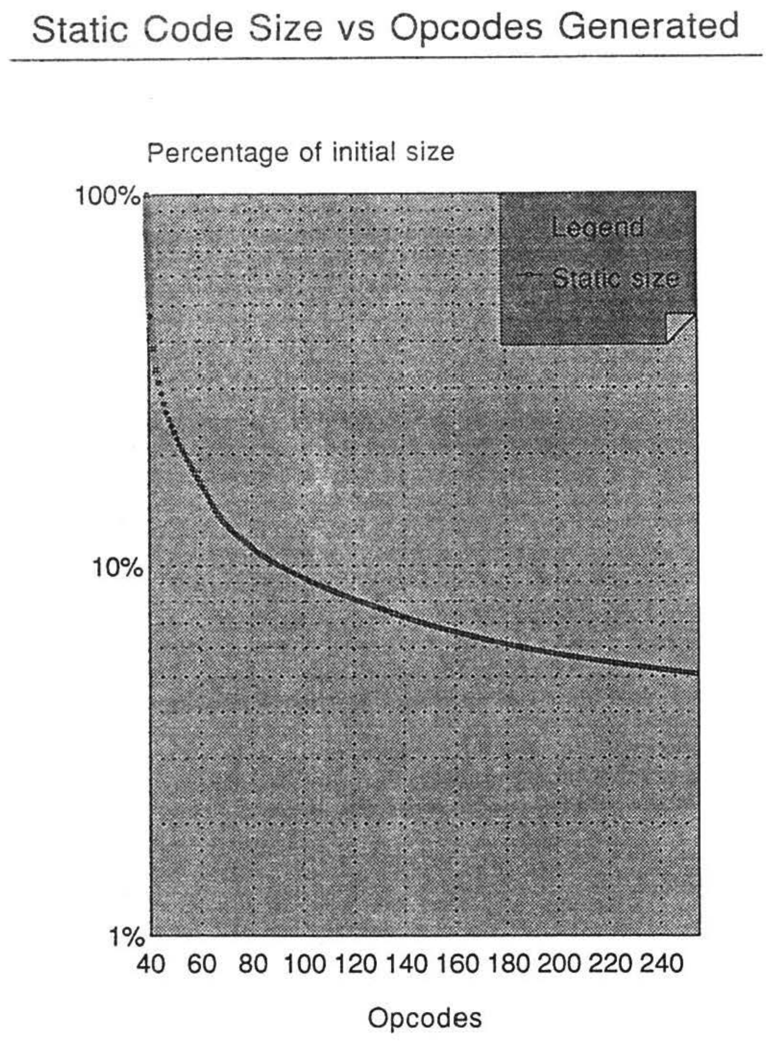

Figure 1: *It is the first few instructions generated that contribute towards most of the savings*

Plotting code size against opcode number (figure 1) actually shows that it is the first few opcodes that get generated which are responsible for the biggest savings in accordance with Bennett and Smith (BS89).

The entropy of the new instruction set has been calculated and amounts to 7.31 bits per symbol. This nearly optimal entropy is a very significant result especially in the light of the fact that only 230 (7.4 bits) out of the 256 opcodes generated are used. Entropy calculations and instruction frequencies can be found in Appendix C.

5 Improving ISGEN

ISGEN uses a greedy algorithm to generate new instructions thus assuming successive substitutions to be independent. This is not necessarily the case and choosing the transformation that leads to the best saving at each step, i.e. locally optimizing a construct, may not achieve a globally optimal encoding. To solve this problem some form of lookahead or a different optimization technique need to be adopted.

Genetic Algorithms (Hol75) have been proved to be a valid optimization technique especially when the final goal is robustness: getting the right balance between efficiency and efficacy that allows to survive in many different environments. The need for robustness is very much felt when designing instruction sets. Automated instruction sets design tools face the challenge of balancing a variety of architectural features, the fine tuning of which is critical to the performance of a fast, economical computer which will run efficiently a wide range of applications. Our goal is to exploit the principles of genetics and the techniques of genetic algorithms to evolve robust, close to optimality, efficient processors from a basic functional specification.

Work is being carried out to develop suitable operators that allow to hybridise the principles of ISGEN with the techniques of Genetic Algorithms. In this hybird methodology a complete compiling instruction set is considered to be a single chromosome. Genes are made up of nucleotides, which are defined as possible instructions obtained under design rules from a canonical genotype. Evolutionary pressure applied to an initial population of pseudo randomly generated chromosomes yields better and better instruction sets.

The new program will be able to cope in a straightforward way with different styles of instruction sets and it will be easy to adjust to optimise different features. Methodology used and performance of ISGEN-GA are the subject of a forthcoming paper.

6 Conclusions

- We have presented a fully automated tool capable of generating a nearly optimal instruction set from a basic specification.
- We have showed its efficacy in optimizing not only artificially verbose canonical instruction sets, but real ones as well.
- Automated instruction set design tools have the following advantages:
  - they assist designing processors that make the most efficient possible use of their resources: different features can be finely tuned at one time (e.g. static program size, bus load, instruction set size, register set size).
  - they reduce the design time: the designer is only required to outline the most general operations a machine should be able to perform and provide a compiler from the source language to this basic set.
  - they provide objective measures of optimality for the generated instruction set.

- instructions are added or removed from a set according to objective efficiency criteria, not according to what the designer feels ought to be there or not.

7 Acknowledgements

We would like to thank the members of the Computing Group at Bath University and especially the EuLISP project team who have provided the environment in which this work could be carried out and Chris Burdorf and Pete Broadbery for their careful reading and comments.

Francesco ARCI and Jeremy BENNETT were both supported by research studentships from the Science and Engineering Research Council of Great Britain during the period in which this work was carried out.

8 Bibliography

References

(Abr63) N. Abramsom. *Information Theory and Coding.* McGraw Hill, 1963.

(AK84) F.J. Ayala and J.A.Jr. Kiger. *Modern Genetics.* The Benjamin/Cummings Publishing Company, Inc., 1984.

(Ben88) J.P. Bennett. *A Methodology for Automated Design of Computer Instruction Sets.* PhD thesis, University of Cambridge, 1988.

(BS89) J.P. Bennett and G.C. Smith. The need for reduced byte stream intruction sets. *The Computer Journal*, 32:370–373, April 1989.

(Dav91) L. Davis, editor. *Handbook of Genetic Algorithms.* Van Nostrand, Reinhold, 1991.

(Gol89) D.E. Golberg. *Genetic Algorithms in search, optimization and machine learning.* Addison-Wesley, 1989.

(Hol75) J.H. Holland. *Adaptation in natural and artificial systems.* The University of Michigan Press, 1975.

(Mat85) D.C.J. Matthews. Poly manual. Technical Report 63, Cambridge University Computer Laboratory, 1985.

(PD80) D.A. Patterson and D.R. Ditzel. The case for the reduced instruction set computer. *Computer Architecture News*, 6:25–33, August 1980.

(PN+90) J. Padget, G. Nuyens, et al. The eulisp definition version 0.69. Technical report, University of Bath, 1990.

(RWS80) M. Richards and C. Whitby-Strevens. *BCPL - The language and its compiler.* Cambridge University Press, 1980.

(SS82) R.E. Sweet and J.G. Sandman. Empirical analysis of the mesa instruction set. In *Proceedings of the ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 235–243, March 1982.

(Tan78) A.S. Tanenbaum. Implications of structured programming for machine architecture. *Communications of the ACM*, 21:237–246, March 1978.

(vdG89) A.J. van de Goor. *Computer Architecture and design.* Addison Wesley, 1989.

9 Appendix A: a canonical instruction set for BCPL

Instruction	Description
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 $a_1, ..., a_n$	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. Unconditionally branch forward a_1.
LOOP a_1	Jump to end of loop. Unconditionally branch forward a_1. Really needs LOOPBACK as well for efficiency.
RESULTIS a_1	Unconditionally jump forward a_1 out of VALOF block.
ENDCASE a_1	Unconditionally jump forward a_1 out of SWITCH block.
RETURN a_1	Return from procedure. Absolute branch address given in stack frame.
GOTO	Jump to absolute address on top of internal stack
FINISH	Terminate program

10 Appendix B: canonical BEEP

Number	*Opcode name*	*Description*
0	UNKNOWN	Opcode 0 is left unused.
1	PUSH-CONSTANT arg	Push a word sized argument of the stack.
2	PUSH-REG	Pop register class and register number. Push selected register.
3	POP-REG	Pop register class and register number. Pop stack into selected register.
4	PUSH-DISPLAY	Pop frame number, pop offset, push display(frame, offset).
5	POP-DISPLAY	Pop frame number, pop offset, pop stack into display(frame,offset).
6	BRANCH	Pop condition result, pop destination. If condition result is non nil then jump to destination.
7	JUMP	Pop destination. Jump to destination.
8	CALL-IN-CURRENT-MODULE	Pop function number, call function.
9	CALL-IN-OTHER-MODULE	Pop module name, pop function number, call function.
10	RETURN	Return from function call.
11	CONS	Pop car, pop cdr, push a cons cell containing them.
12	GCTRAP	Pop *type*, pop *items-no*. Trap to garbage collector if allocation of *items-no* items of type *type* would cause a garbage collection.
13	EQ	Pop *arg1*, pop *arg2*, push (eq *arg1 arg2*).
14	PUSH-STATIC	Pop static vector index. Push value at index on stack.
15	CAR	Pop *arg*, push (car *arg1*).
16	CDR	Pop *arg*, push (cdr *arg1*).
17	PUSH-NON-LOCAL-VALUE	Pop module name, pop *index*, push value at *index* in values vector of specified module on stack.
18	PUSH-LOCAL-VALUE	Pop *index*, push value at *index* on stack.
19	POP-LOCAL-VALUE	Pop *index*, pop stack into values vector at index *index*
20	PUSH-INTERNAL-FUNCTION	Pop function number, push function object on stack.

Number	Opcode name	Description
21	BEGIN-W-CC	Install the function on top of stack as the current continuation and save previous.
22	END-W-CC	Deinstall current continuation.
23	APPLY	Apply function on top of stack to arguments on *top-1* of stack.
24	BEGIN-W-H	Install function on top of stack as the current handler and save previous.
25	END-W-H	Restore the previous handler.
26	ALLOC	Allocate a display frame capable of holding top of stack lisp objects.
27	DEALLOC	Deallocate current display frame.
28	CALL-SELF	Call function tail recursively.
29	RPLACA	Pop a lisp object from stack and push back a cons cell holding the item just popped in the car field.
30	RPLACD	Pop a lisp object from stack and push back a cons cell holding the item just popped in the cdr field.
31	BEGIN-U-P	Save function on top of stack as the current cleanup pointer, saving previous
32	END-U-P	Restore previous cleanup pointer.
33	PUSH-DYNAMIC	Push dynamic variable specified by value on top of stack on the stack.
34	POP-DYNAMIC	Pop dynamic variable number, pop value, set variable to value just popped.
35	BIND	Create fresh dynamically scoped bindings for identifier on stack and initialise the binding to *top-1* of stack.
36	UNBIND	Restore the previous dynamic binding of the identifier on top of stack. Unbinding must be in opposite order to binding.
37	VECTOR-REF	Pop index of element to access, pop vector, push required element.
38	UPDATE-VECTOR-REF	Pop index of element to access, pop vector, pop element and store it in vector.
39	ALLOC-IREGS	Pop and set number of input registers used by function.
40	ALLOC-LREGS	Pop and set number of local registers used by function.

11 Appendix C: Entropy in BEEP

Opcode	Occurrences	Frequency	Information content
0	0	0.000000	0.00
1	4	0.000284	11.78
2	140	0.009933	6.65
3	36	0.002554	8.61
4	105	0.007450	7.07
5	110	0.007805	7.00
6	0	0.000000	0.00
7	2	0.000142	12.78
8	0	0.000000	0.00
9	0	0.000000	0.00
10	0	0.000000	0.00
11	21	0.001490	9.39
12	0	0.000000	0.00
13	0	0.000000	0.00
14	31	0.002200	8.83
15	0	0.000000	0.00
16	0	0.000000	0.00
17	22	0.001561	9.32
18	15	0.001064	9.88
19	14	0.000993	9.98
20	3	0.000213	12.20
21	4	0.000284	11.78
22	4	0.000284	11.78
23	0	0.000000	0.00
24	3	0.000213	12.20
25	3	0.000213	12.20
26	10	0.000710	10.46
27	0	0.000000	0.00
28	21	0.001490	9.39
29	2	0.000142	12.78
30	4	0.000284	11.78
31	0	0.000000	0.00
32	0	0.000000	0.00
33	6	0.000426	11.20
34	3	0.000213	12.20
35	3	0.000213	12.20
36	3	0.000213	12.20
37	4	0.000284	11.78
38	1	0.000071	12.78
39	7	0.000497	10.98
40	4	0.000284	11.78
41	40	0.002838	8.46
42	26	0.001845	9.08
43	54	0.003831	8.03
44	61	0.004328	7.85
45	99	0.007024	7.15
46	33	0.002341	8.74
47	123	0.008727	6.84
48	33	0.002341	8.74
49	30	0.002129	8.88
50	60	0.004257	7.88
51	82	0.005818	7.41
52	85	0.006031	7.37
53	26	0.001845	9.08
54	111	0.007876	6.99
55	0	0.000000	0.00
56	27	0.001916	9.03
57	0	0.000000	0.00
58	58	0.004115	7.92
59	96	0.006811	7.20
60	9	0.000639	10.61
61	78	0.005534	7.50
62	229	0.016248	5.94
63	59	0.004186	7.90
64	0	0.000000	0.00
65	74	0.005250	7.57
66	115	0.008160	6.94
67	134	0.009508	6.72
68	72	0.005109	7.61
69	0	0.000000	0.00
70	102	0.007237	7.11
71	39	0.002767	8.50
72	70	0.004967	7.65
73	105	0.007450	7.07
74	45	0.003193	8.29
75	58	0.004115	7.92
76	183	0.012984	6.27
77	124	0.008840	6.81
78	109	0.007734	7.08
79	177	0.012559	6.32
80	115	0.008160	6.94
81	82	0.005818	7.43
82	72	0.005109	7.61
83	64	0.004541	7.78
84	176	0.012488	6.32
85	92	0.006528	7.26
86	0	0.000000	0.00
87	18	0.001277	9.61
88	338	0.023982	5.38
89	0	0.000000	0.00
90	4	0.000284	11.78
91	174	0.012346	6.34
92	29	0.002058	8.92
93	84	0.005960	7.39
94	216	0.015326	6.03
95	212	0.015042	6.05
96	30	0.002129	8.88
97	168	0.011920	6.39
98	28	0.001987	8.98
99	42	0.002980	8.39
100	212	0.015042	6.05
101	113	0.008018	6.96
102	22	0.001561	9.32
103	83	0.005889	7.41
104	0	0.000000	0.00
105	0	0.000000	0.00
106	2	0.000142	12.78
107	146	0.010359	6.59
108	50	0.003548	8.14
109	217	0.015397	6.02
110	0	0.000000	0.00
111	2	0.000142	12.78
112	7	0.000497	10.98
113	0	0.000000	0.00
114	5	0.000355	11.46
115	108	0.007663	7.03
116	41	0.002909	8.43
117	24	0.001703	9.20
118	123	0.008727	6.84
119	219	0.015539	6.01
120	12	0.000851	10.20
121	49	0.003477	8.17
122	61	0.004328	7.85
123	21	0.001490	9.39
124	80	0.005676	7.46
125	78	0.005534	7.50
126	72	0.005109	7.61
127	5	0.000355	11.46
128	20	0.001419	9.46
129	113	0.008016	6.96
130	76	0.005392	7.53

Opcode	Occurrences	Frequency	Information content
131	13	0.000922	10.08
132	9	0.000639	10.61
133	12	0.000851	10.20
134	21	0.001490	9.39
135	94	0.006670	7.23
136	126	0.008940	6.81
137	69	0.004896	7.67
138	44	0.003122	8.32
139	30	0.002129	8.88
140	4	0.000284	11.78
141	23	0.001632	9.26
142	44	0.003122	8.32
143	54	0.003831	8.03
144	159	0.011281	6.47
145	117	0.008301	6.91
146	81	0.005747	7.44
147	22	0.001561	9.32
148	90	0.006386	7.29
149	86	0.006102	7.36
150	78	0.005534	7.50
151	137	0.009720	6.68
152	137	0.009720	6.68
153	76	0.005392	7.53
154	77	0.005463	7.52
155	70	0.004967	7.65
156	2	0.000142	12.78
157	122	0.008656	6.85
158	12	0.000851	10.20
159	66	0.004683	7.74
160	126	0.008940	6.81
161	20	0.001419	9.46
162	63	0.004470	7.81
163	0	0.000000	0.00
164	115	0.008160	6.94
165	111	0.007876	6.99
166	55	0.003902	8.00
167	66	0.004683	7.74
168	109	0.007734	7.01
169	108	0.007663	7.03
170	59	0.004186	7.90
171	43	0.003051	8.36
172	40	0.002838	8.46
173	101	0.007166	7.12
174	39	0.002767	8.50
175	99	0.007024	7.15
176	96	0.006811	7.20
177	96	0.006811	7.20
178	62	0.004399	7.83
179	94	0.006670	7.23
180	17	0.001206	9.70
181	28	0.001987	8.98
182	0	0.000000	0.00
183	44	0.003122	8.32
184	85	0.006031	7.37
185	29	0.002058	8.92
186	31	0.002200	8.83
187	94	0.006670	7.23
188	27	0.001916	9.03
189	33	0.002341	8.74
190	9	0.000639	10.61
191	82	0.005818	7.43
192	81	0.005747	7.44
193	27	0.001916	9.03
194	11	0.000780	10.32
195	79	0.005605	7.48
196	75	0.005321	7.56
197	12	0.000851	10.20
198	30	0.002129	8.88
199	11	0.000780	10.32
200	69	0.004896	7.67
201	68	0.004825	7.70
202	67	0.004754	7.72
203	67	0.004754	7.72
204	23	0.001632	9.26
205	64	0.004541	7.78
206	63	0.004470	7.81
207	63	0.004470	7.81
208	63	0.004470	7.81
209	62	0.004399	7.83
210	19	0.001348	9.53
211	61	0.004328	7.85
212	61	0.004328	7.85
213	61	0.004328	7.85
214	60	0.004257	7.88
215	60	0.004257	7.88
216	60	0.004257	7.88
217	59	0.004186	7.90
218	58	0.004115	7.92
219	57	0.004044	7.95
220	56	0.003973	7.98
221	55	0.003902	8.00
222	10	0.000710	10.46
223	54	0.003831	8.03
224	4	0.000284	11.78
225	54	0.003831	8.03
226	8	0.000568	10.78
227	25	0.001774	9.14
228	50	0.003548	8.14
229	50	0.003548	8.14
230	50	0.003548	8.14
231	50	0.003548	8.14
232	49	0.003477	8.17
233	49	0.003477	8.17
234	48	0.003406	8.20
235	47	0.003335	8.23
236	47	0.003335	8.23
237	2	0.000142	12.78
238	45	0.003193	8.29
239	45	0.003193	8.29
240	44	0.003122	8.32
241	44	0.003122	8.32
242	0	0.000000	0.00
243	44	0.003122	8.32
244	44	0.003122	8.32
245	44	0.003122	8.32
246	2	0.000142	12.78
247	43	0.003051	8.36
248	43	0.003051	8.36
249	27	0.001916	9.03
250	42	0.002980	8.39
251	42	0.002980	8.39
252	42	0.002980	8.39
253	42	0.002980	8.39
254	41	0.002909	8.43
255	41	0.002909	8.43
256	40	0.002838	8.46

Total	14094
Entropy:	7.33
Unused opcodes:	25
Total used opcodes:	232