The Need for Reduced Byte Stream Instruction Sets

J. P. BENNETT* AND G. C. SMITH
School of Mathematical Sciences, University of Bath, Claverton Down, Bath BA2 7AY

In the design of byte stream instruction sets, a popular methodology has been peephole refinement of a canonical instruction set. By this we mean the addition of extra opcodes over the minimum necessary for a viable machine in order to handle particularly common cases more efficiently. Such machines by the use of 'escape' sequences often have hundreds of different opcodes. By comparison, modern 'Reduced Instruction Set' computers, which do not take the byte stream approach, have a very small number of opcodes. We argue that in fact there is no significant benefit to be achieved by having byte stream instruction sets of more than around 100 opcodes. A formal basis for the selection of extra opcodes is used in justifying this view.

Received May 1987, revised September 1987

1. INTRODUCTION

Many computer architectures and intermediate codes, both general and specialised, are based on the concept of a byte stream instruction set. Instructions in such a machine consist of a single-byte opcode specifying the required operation, possibly followed by a number of argument bytes.† The number and format of such arguments is implied by the opcode itself. The number of opcodes in such architectures is limited by the size of a byte. Today a byte is almost invariably eight bits, which means there may be up to 256 opcodes. This may be extended arbitrarily by selecting some of the 256 opcodes to be 'escape' opcodes, with a subsequent byte acting as a secondary opcode. A typical example of this is EM-1.[8]

A common manner of designing such instruction sets is to select a range of opcodes sufficient to support the major concepts embodied in the architecture. We refer to this as the 'canonical instruction set'. Supplementary opcodes, up to the maximum 256 (or more if there are escape sequences) are then added that support particularly common cases in the use of the canonical instruction set. Typically these extra opcodes are derivatives of existing opcodes. Thus an architecture may provide a canonical LOAD NUMBER instruction, being a one-byte opcode and four-byte argument, to load a four-byte constant, within the canonical instruction set. Extra instructions may then be added to deal with loading small constants (LOAD NUMBER BYTE, one-byte opcode, one-byte argument) or specific values (LOAD NUMBER ZERO, one-byte opcode, no argument) for example. Furthermore common opcode pairs may be combined to generate a single instruction, for example PLUS and LOAD NUMBER may create ADD NUMBER, to add a constant. These rules for deriving new instructions may be applied repetitively, so we may suggest an opcode ADD NUMBER ONE, to add the constant one.

Such instructions improve the instruction set by reducing the space occupied by compiled code and reducing execution time required for fetching of arguments. It is important to have suitable quantifiable design criteria, such as code size and execution speed, against which the benefit accruing from adding a particular opcode may be measured. This methodology of instruction set design, and its automation is reviewed and discussed in detail by Bennett.[2]

This paper looks at whether the number of special opcodes to support a particular canonical opcode or group of opcodes can be quantified. There is no reference to such quantification in the literature; all work in this field relies on the experience and intuition of the instruction set designer when choosing opcodes. The only opinion that could be found on the subject was with regard to CINTCODE, a byte stream instruction set which supports compiled BCPL,[7] with the aim of minimising compiled code size.[6] Richards suggests[5] that the number of extra instructions added to support a particular canonical opcode or group of opcodes should be proportional to the frequency with which that opcode or group of opcodes is found in compiled canonical code. Thus the observation that half the instructions in compiled canonical code are to do with loading and storing leads to the suggestion that half the 256 opcodes should be load and store opcodes. Furthermore, the observation that loading is twice as common as storing leads to the suggestion that the 128 load and store opcodes should be chosen in a ratio of 2 to 1. CINTCODE does indeed follow this general idea, although the data on the frequency of instructions in compiled code are based more on the designer's (considerable) experience than detailed statistical analysis.

We suggest that such an approach is not completely correct. Specifically we suggest that there is little benefit to be gained from adding more than a few special opcodes to the canonical instruction set. There is no need for a full complement of 256 opcodes, let alone the use of additional escape opcodes. We first present a justification of this design approach and then quantify the benefit that accrues as we add additional opcodes. This is illustrated by two practical examples of instruction set design.

2. THE INTEGER PROGRAMMING APPROACH

Let us suppose that our canonical opcodes are partitioned into t types, $T_1, T_2, \ldots, T_t$. This may for example be at the level of partitioning into data access, data manipulation

* To whom correspondence should be addressed.

† The terms 'opcode' and 'instruction' are used very loosely throughout the literature. We shall use 'opcode' to mean the initial byte specifying the operation required, whilst 'instruction' will be used to refer to the opcode together with its requisite number of argument bytes.

and flow of control opcodes ($t = 3$), or it may be at the level of the canonical opcode, with initially only one opcode of each type, and t possibly quite large. We also assume that we have a quantifiable criterion for measuring the benefit accruing though adding extra opcodes of a particular type. This may be the percentage saving in code size, or the decrease in execution time per instruction observed when adding extra opcodes of a particular type. We now construct our full instruction set by adding a number of extra opcodes to support each particular type. For each type T_i let there be x_i extra opcodes. In any body of compiled code we observe that opcodes of type T_i occur with frequency f_i. Clearly the higher this frequency the more incentive there is to choose x_i to be large, since then we increase the benefit due to these opcodes. For any given type, T_i, adding extra opcodes will give a benefit s_i, given by some function g_i of x_i

$$s_i = g_i(x_i).$$

And thus the total benefit will be

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

We wish to maximise S subject to the constraint $\Sigma_1^t x_i = c$, i.e. there is a limit on the number of extra opcodes permitted, typically 256 less the number of canonical opcodes. This is a standard problem in integer programming, the stock-cutting problem. In the case of instruction set design we observe that if we define

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

then $\Delta s_{i,j-1} \leqslant \Delta s_{i,j}$ for all $i \in 1, \ldots, t$ and $j > 1$. This makes the problem particularly easy to solve by a simple 'greedy' algorithm of adding one extra opcode, the best available, at a time until we have reached the constraint, i.e. we have c extra opcodes.

This works well as a methodology of instruction set design, although deriving the functions s_i may involve much computer time, typically exhaustive enumeration of all possible selections of opcodes. In addition it is not always strictly true that $\Delta s_{i,j-1} \leqslant \Delta s_{i,j}$, although the observations in Ref. 2 suggest that in practice it is very close to the truth.

3. CASE STUDIES

An instruction set generator, ISGEN, has been developed,[1] which uses a greedy algorithm to add instructions to a canonical instruction set. In our first example it was used to add instructions to a canonical instruction set of 49 instructions supporting 32-bit BCPL, on the basis of the frequency of instructions in 500K of compiled code. In the second example ISGEN was used to add instructions to an existing instruction set of 26 instructions supporting the programming language POLY[3] on the basis of the frequency of intructions in 200K of compiled code. In both cases the primary design criterion was reduction in static size of compiled code. The effect of adding instructions was followed by looking at the space occupied by the sample body of code as each new instruction was incorporated by peephole substitution.

This is satisfactory for instruction-set design research, but is heavy on computer resources. At least initially it would be helpful if the computer design could be given a guide as to the number of opcodes required to support each type of instruction. To this end a model of the benefits that accrue through the addition of opcodes is desirable. It is in the construction of such a model that we see where the underlying weakness of this design method lies.

4. A MATHEMATICAL MODEL

There are various ways of making a mathematical model of benefit that will accrue for different values of x_i. Let us suppose we can approximate the benefit functions g_i by a decaying exponential.

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

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

Thus the total benefit is:

$$S = \sigma \sum_{i=1}^{t} f_i(1 - e^{-a_i x_i}) \tag{1}$$

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

We wish to find values of x_i which will maximise S. To this end we use the method of Lagrange multipliers. The reader is referred to any elementary analysis text for an explanation.

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

Let $\psi = S + \lambda g$.

We must solve simultaneously

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

Thus

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

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

$$\prod_{i=1}^{t} (a_k f_k e^{-a_k x_k})^{d_i} = \prod_{i=1}^{t} (a_i f_i e^{-a_i x_i})^{d_i}.$$

Therefore

$$(a_k f_k e^{-a_k x_k})^{\Sigma_{i=1}^{t} d_i} = \prod_{i=1}^{t} (a_i f_i)^{d_i} e^{-a \Sigma_{i=1}^{t} x_i},$$

where

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

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

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

Now we have

$$(a_k f_k e^{-a_k x_k})^b = \prod_{i=1}^{t} (a_i f_i)^{d_i} e^{-ac}.$$

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

$$\eta_k e^{-ba_k x_k} = r$$

and finally we have

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

Such a choice of x_k's will maximise S.

The method of Lagrange multipliers ensures that we have found a critical point of the saving function. By considering the problem in question we deduce that this is a maximum.

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

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

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

and

$$\log_e(r) = \log_e\left(\prod_{i=1}^{t} (a_i f_i)^{d_i} e^{-ac}\right)$$
$$= \sum_{i=1}^{t} d_i \log_e(a_i f_i) - ac.$$

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

5. EXPERIMENTAL SUPPORT

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

We tried fitting Equation (1) to our data, to obtain values of a_i, f_i and hence obtain predictions of the number of instructions of each type to be incorporated in the instruction set. There is a slight problem in that instructions formed by combination of instructions of two different types end up belonging to a new sort of combined type (for example CALLGLOBAL), but this problem is overcome by counting such an instruction as half an instruction of one type and half of the other.

Table 1. Model instruction set distribution for three types.

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

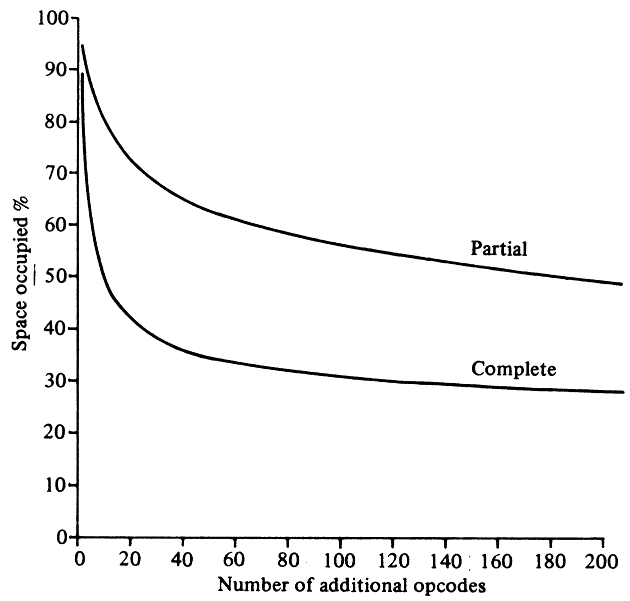

Figure 1. Percentage space saving due to additional opcodes.

Statistically, fitting Equation (1) by least-squares regression is far from satisfactory, particularly since we have no model of error behaviour. However, Table 1 shows the best results that we could achieve using our model Equation (1). The prediction of number of opcodes is not too bad; ISGEN suggests that a partition of 137, 10 and 60 is optimal. However, the predicted saving of 72400 bytes is out by a factor of nearly 5 from the saving of 358762 bytes in the size of the code sample achieved with the instruction set selected by ISGEN.

The model of Equation (1) would appear inadequate for the task. It is unable to give really reliable predictions of the number of opcodes of different types required. Its predictions of the savings to be achieved bear no discernible relationship to reality. To be of any use in practice we would not measure σf_i and a_i by curve fitting of completed data, but would attempt to estimate them in advance. This would of course lead to even less believable predictions.

These results have led us to review the mechanism of this style of instruction set design.

6. A NEW CRITERION FOR OPCODE SELECTION

The source of the problem with our model can be seen if we look at the savings that accrue as we add opcodes with ISGEN. A graph of space occupied by the sample compiled code as a percentage of its initial size as we incorporate in turn each additional opcode suggested by ISGEN is given in Fig. 1. For comparison we show a partial use of ISGEN, permitting only the option of generating an opcode by combination with a specific argument value, rather than the three ways of generating outlined above; roughly akin to the methodology of Tanenbaum.[8] The fact is that whether we use 110 or 140 opcodes of type 1, or 10 or 20 of type 2 or 60 or 70 of type 3 makes very little difference. Almost all our benefit is coming from the first few special opcodes that we add. When we run ISGEN using three methods of generating new opcodes, 90% of the space saved is achieved from the first 44 additional opcodes of all types. Thereafter additional opcodes make very little difference. Even with our partial use, after the manner of Tanenbaum we see 90% saving using little more than half the additional opcodes. When we look at our model we see that this is where we are on the flat part of the exponential decay. We can also explain the error in estimation of savings. This is due to error in fitting the first rapidly decaying part of the exponential. Slight errors in estimation of a_i will lead to gross under- or over-estimates of the savings to be expected. Our efforts to refine to the last opcode the addition of opcodes to our instruction set are really pointless. We can achieve 90% of our aim with an instruction set of just 93 opcodes. If we look at less easily quantified measures of benefit, such as dynamic code size or memory bus loading, we are unlikely to be able to quantify the benefit due to a particular opcode with as much as 10% accuracy anyway. ISGEN has in any case achieved a 17% improvement merely by extending from one to three the number of methods of constructing new opcodes.

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

7. CONCLUSIONS

It seems there is a simple message for the designer of a byte stream architecture. Only add those extra opcodes you really need. There really is little point in filling up an instruction set with little-used opcodes. Fewer opcodes mean less microcode, and more silicon for hardware assistance. It also makes the compiler writer's job easier, with fewer options in selecting opcodes. Under these conditions many of the benefits seen from RISC technology (see for example Ref. 4) may be possible, whilst retaining the advantages found with byte stream architectures.

REFERENCES

1. J. P. Bennett, *Automated Design of an Instruction Set for BCPL.* Technical Report No. 93, University of Cambridge Computer Laboratory (1986).
2. J. P. Bennett, A Methodology for Automated Design of Computer Instruction Sets. *Ph.D. Thesis*, University of Cambridge (1987).
3. D. C. J. Matthews, *Poly Manual.* Technical Report No. 63, University of Cambridge Computer Laboratory (1985).
4. D. A. Patterson and C. H. Sequin, A VLSI RISC. *Computer* **15** (9), 8–21 (1982).
5. M. Richards, The Design of CINTCODE (personal communication, 1983).
6. J. Richards and C. Jobson, *BCPL for the BBC Microcomputer.* Acornsoft Ltd, Cambridge (1982).
7. M. Richards and C. Whitby-Strevens, *BCPL, the Language and its Compiler.* Cambridge: Cambridge University Press (1980).
8. A. S. Tanenbaum, Implications of structured programming for machine architecture. *Communications of the ACM* **21** (3), 237–246 (1978).