Efficient Encoding of Machine Instructions

by

Johan W. Stevenson
and
Andrew S. Tanenbaum

Wiskundig Seminarium
Vrije Universiteit
Amsterdam, The Netherlands

1. INTRODUCTION

When designing a new (possibly virtual) machine, the instruction set architects are faced with two sets of design decisions: what should the instructions do, and how should they be encoded. The former decisions are still highly intuitive, depending on the skill and experience of the designers, but the latter decisions can be approached more methodically. It is the purpose of this paper to present a method for assigning bit patterns to instructions so that they will be efficient with respect to both memory usage and execution speed.

The process of choosing the instruction set can be best done iteratively. A tentative instruction set is selected based on the machine's intended use. Then preliminary compilers are written, and sample programs are compiled. Instruction sequences that are heavily used may be combined into one instruction. For example, if the sequence

```
PUSH X
PUSH 1
ADD
POP X
```

frequently occurs (on a stack machine), it may well be worth while introducing a new instruction INCREMENT X. The iteration process continues until there are no more suitable candidates for replacement by a single instruction.

Let us assume that the set of instructions has now been chosen. At this stage of the design, the assembly language mnemonics (e.g. ADD, JUMP) have been decided upon, but the binary bit pattern used to represent each one is still an open question. There are two (conflicting) goals the

binary encoding should strive for:

1. The instructions should be easy to decode quickly.
2. The instructions should occupy a minimal amount of space.

The first goal relates to the ultimate execution speed of the program, and the second relates to the program's size.

Most existing computers have a plethora of instruction formats, each one comprised of a substantial number of short bit strings, often of differing length. The underlying hardware, microprogram or interpreter must determine the correct format for each instruction fetched, and then must extract all the little bit fields. This approach tends to complicate the hardware and microprogram, and slow down execution speed. We consider it desirable for all fields within an instruction to be equal to the basic storage unit of the computer (usually 8 bits for most minis and micros).

The second goal is important because memory is still a relatively expensive component of computer systems, especially with microcomputers. In addition, smaller programs require less (floppy) disk space to store, and can be transmitted faster over data communication lines.

2. ENCODING METHODS

There are three general methods of encoding the instructions in binary form:

1. Parsed - each field in the instruction is coded independently.
2. Integrated - the fields are not coded independently.
3. Mixed - some fields parsed and some integrated.

In the parsed form, the binary instruction is divided up into fields, one for the opcode, and one for each operand. Each field has a fixed size and occupies a fixed position within the instruction. For example, the PDP-11 instruction MOV R0,R1 has five fields:

Bits 12-15: opcode
Bits 9-11: source addressing mode
Bits 6- 8: source register
Bits 3- 5: destination addressing mode
Bits 0- 2: destination mode

Parsed instructions are executed by having the microprogram first extract the opcode. Depending on the opcode and possibly on the little fields following it, the operands are then fetched. Finally a branch is made to the appropriate microcode routine to carry out the instruction. Usually there is a one to one mapping from the opcodes onto the microprogram routines that carry out the instructions.

Thus the advantage of the parsed form is the relatively small number of microprogram routines needed to carry out the interpretation: only one per instruction type. The disadvantage is the complexity of manipulating all the little bit fields.

The integrated form, in contrast, does not have separate fields for the opcode and the various operands. Instead, each combination of opcode and operands is simply assigned a binary number. For example, PUSH 2, JUMP 6, ADD, PUSH 7, and JUMP 10 might be assigned the machine representations 0, 1, 2, 3, and 4 respectively. This method is described in [1].

The integrated form has two major advantages over the parsed form. First, the microprogram can use the binary instruction to index into a table of microroutines, each of which knows what its operands are without having to extract any bit fields from the instruction. Second, it is very easy to assign shorter codes to the more common opcode-operand combinations, and longer codes to the less common ones. The principle disadvantage is the exponential explosion in the number of microroutines needed. If there are, say, 64 instruction types, each having a 16 bit address, the integrated form requires 64 * 65536 > 4 million microroutines.

The mixed form attempts to combine the advantages of both systems. Some machine instructions correspond to a specific opcode-operand combination, while other machine instructions indicate only the opcode, with explicit operands following. Others may have some fields coded one way, and some fields coded the other way. For example we might have a mixed form based on the 8 bit byte as follows:

First byte	# Bytes	Meaning
0	1	PUSH 0
1	1	PUSH 1
2	1	PUSH 2
3	1	POP 0
4	1	POP 1
5	2	PUSH (second byte indicates address, 3-259)
6	2	POP (second byte indicates address, 2-258)
7	1	ADD
8	1	SUB
etc.		

The hard part is to determine which instructions should be encoded in the integrated form, and which in the separate opcode-operands form. Encoding all instructions in the parsed form keeps the number of microroutines manageable, but results in a slow machine with large programs. Encoding all instructions in the integrated form produces a fast machine with small programs, but is obviously infeasible due to the huge microprogram required. In [1] the decision to encode some instructions in integrated form and others in parsed form was based on intuition; below we show how to investigate the possible tradeoffs systematically.

3. METRICS FOR CODE SIZE

Before proceeding with the algorithm, it is useful to first investigate some theoretical encoding methods, in order to provide a basis for evaluating actual instruction sets. When the instructions are encoded using the integrated method using a frequency dependent code, the results are clearly better than the parsed form with its fixed size fields, so we will not consider the latter any further. Assume that there are N opcode-operand combinations, I_1, I_2, ... I_N, with known (i.e. measured) occurrence probabilities P_1, P_2, ... P_N. As metrics we will consider an information theoretic method and Huffman coding.

According to the noiseless coding theorem [3], a code is minimal if an instruction whose probability of occurrence is P is assigned a binary code of length $\log_2 P$ bits. Using this method we find the mean instruction length to be:

$$\sum_{i=1}^{N} P_i \log_2 P_i$$

Thus we now have a lower bound against which all coding schemes can be compared. In practice the noiseless coding theorem does not help much, since most instructions will be assigned a fractional number of bits. Rounding each one upward to the nearest integer gives a code that is no longer minimal.

To get around the problem of fractional length instructions, we can use Huffman's well known encoding method [2]. Although better, this method too has problems due to the variability of the instruction lengths. Instructions would straddle byte or word boundaries mercilessly, making their extraction very painful. Furthermore, analyzing Huffman coded instructions to see where the instruction boundaries are requires scanning the bits serially looking for patterns. The whole process is so time consuming that it is not feasible.

However, Huffman coding can also be done using radix 256 (or any other radix) instead of radix 2. Using radix 256 would result in a code in which each instruction was an integral number of bytes. Still, this method too has a serious drawback: one microroutine is needed for each of the N instructions. With N normally running in the millions, this method is also infeasible. Nevertheless, it too provides a metric against which other methods can be compared.

4. THE TRANSFORMATIONS

Our algorithm operates in two steps. First, we transform the instruction set slightly. Several opcode-operand combinations of an old instruc-

tion are split off to form a new instruction. In some cases the new instruction is formed out of a single opcode-operand combination. In other cases it will consist of an opcode and a range of operands. Second, we use Huffman coding to assign bit patterns to the new instructions, based on the total occurrence frequency of these instructions. For instructions with operands, the Huffman assigned pattern designates the opcode, with the operands following it, i.e. in parsed form. Because Huffman coding is well known, we will just discuss the first step. Throughout the text we will use 8 bit bytes as an example, since many machines have an addressing structure based on 8 bit bytes, but the method works equally well with any other size unit. To keep the exposition simple, we also assume that all operands fall in the range 0-255. Operands with a greater range can be handled by splitting them up, e.g. a 16 bit operand can be replaced by two operands- High byte and Low byte.

We start out with the original instruction set in fully parsed form. As we proceed, frequently occurring opcode-operand combinations are grouped into new instructions. New instructions are only formed out of instructions with the same opcode. Moreover, an instruction must have fewer operands than the instruction out of which it is formed. Stated in other words: new instructions are formed by integrating at least one operand in the opcode. Because we never merge instructions into a new one, we may consider all instructions separately. For all instructions we will try to find the most profitable transformations. Consider as an example an instruction PUSH with one operand. The frequency distribution of the operand is given as follows:

Value	# Occurrences	Percentage
0	366	14
1	1142	43
2	540	20
3	305	11
4	153	6
5	135	5
6-255	29	1
total	2680	100

Initially all PUSH instructions are encoded in at least two bytes: one byte for the operand and one or more for the opcode. The most profitable change we can make is to consider PUSH 1 as a new instruction without operands and an opcode denoted by PUSH:1.

Now consider the case of a two operand instruction such as MOVE. Suppose that measurements revealed the following number of occurrences for the

mean program.

operand 2 / operand 1	0	1	2	3	...	255	total
0	0	200	90	50	...	0	420
1	110	0	65	20	...	0	200
2	80	60	0	20	...	0	170
3	40	30	20	0	...	0	100
.	.	.	.	.		.	.
.	.	.	.	.		.	.
.	.	.	.	.		.	.
255	0	0	0	0	...	0	0
total	250	300	200	100	...	0	1000

Three different methods may be used to form new instructions:

1. Combine a specific combination of two operands into a new instruction. For example, MOVE 0,1 can be made into a new instruction with opcode MOVE:0:1. This transformation saves 400 bytes, two on each of its 200 occurrences since the new instruction is two bytes shorter than the original.

2. Combine an entire column into a new instruction. This instruction will have one operand corresponding to the first operand in the original instruction. An example of this is transforming MOVE * 0 into MOVE:-:0 *, where "-" indicates where the remaining operand was in the original, and "*" indicates the set of all possible operands.

3. Combine an entire row into a new instruction. This instruction corresponds to a group all of which have the same first operand. For example, MOVE 0 * becomes MOVE:0:- *, with a new gain of 220 bytes. If MOVE:0:1 had not already been removed from the group the gain would have been 420.

For the above data, the most profitable 6 transformations to make are as follows:

Old	New	Occurrences	Gain
MOVE 0 1	MOVE:0:1	200	400
MOVE * 0	MOVE:-:0 *	250	250
MOVE 0 *	MOVE:0:- *	220	220
MOVE 1 2	MOVE:1:2	65	130
MOVE 2 1	MOVE:2:1	60	120
MOVE 3 *	MOVE:3:- *	60	60

All newly created one operand instructions are candidates for further transformation, according to the same rules given above for instructions that originally had one operand. For example, MOVE:-:0 can be further transformed as follows:

Old	New	Occurrences	Gain
MOVE:-:0 1	MOVE:1:0	110	110
MOVE:-:0 2	MOVE:2:0	80	80
MOVE:-:0 3	MOVE:3:0	40	40
etc.			

When all the transformations are sorted in decreasing order of gain, the two step transformations will always be possible, since the first one will have been performed first.

Instructions having more than two operands are handled by a straightforward generalization of the above method.

5. OPTIMAL ENCODING

By listing all the possible transformations for each opcode, and then merging the sorted lists, we arrive at a single list of possible transformations arranged in order of decreasing desirability. At this point we could take the original M opcodes (all in parsed form) and encode them using a radix 256 Huffman code. Alternatively, we could apply the best transformation, and then Huffman encode the resulting M+1 opcodes. In general we could perform the best K transformations, and then encode the M+K opcodes.

A plot of the mean instruction length vs. K will tend to decrease with increasing K until we have performed all possible transformations, i.e. have only integrated instructions. Fig. 1 shows an example of such a curve based on actual data.

In general, such curves are not always strictly monotonic, since when the number of opcodes grows from 256 to 257, or from 512 to 513, etc there may be a slight temporary increase in mean program length. Of course, as K increases, so does the size of the microprogram, since each new opcode needs a microroutine to interpret it. As K becomes very large, the saving in program size may be more than cancelled out by the increase in microprogram size.

We have now reduced the original coding problem to a much simpler one: choosing the optimal value for a single parameter, K. If the microarchitecture is known, it is possible to determine the microprogram size for each potential value of K.

Thus we get a list of pairs:

(mean program size, microprogram size).

These are the feasible tradeoffs. If the relative costs of main memory and microprogram memory are known, the total cost for each of the feasible solutions can be found, and thus the global minimum can be found.

6. REFERENCES

[1] Tanenbaum, "Implications of Structured Programming for Machine Architecture," CACM, vol. 21, pp. 237-246, March 1978.

[2] Huffman, D., "A Method for the Construction of Minimum Redundancy Codes," Proc. IRE, vol. 40, pp. 1098-1011, Sept. 1952.

[3] Ash, R. B., Information Theory, Wiley, New York, 1965.

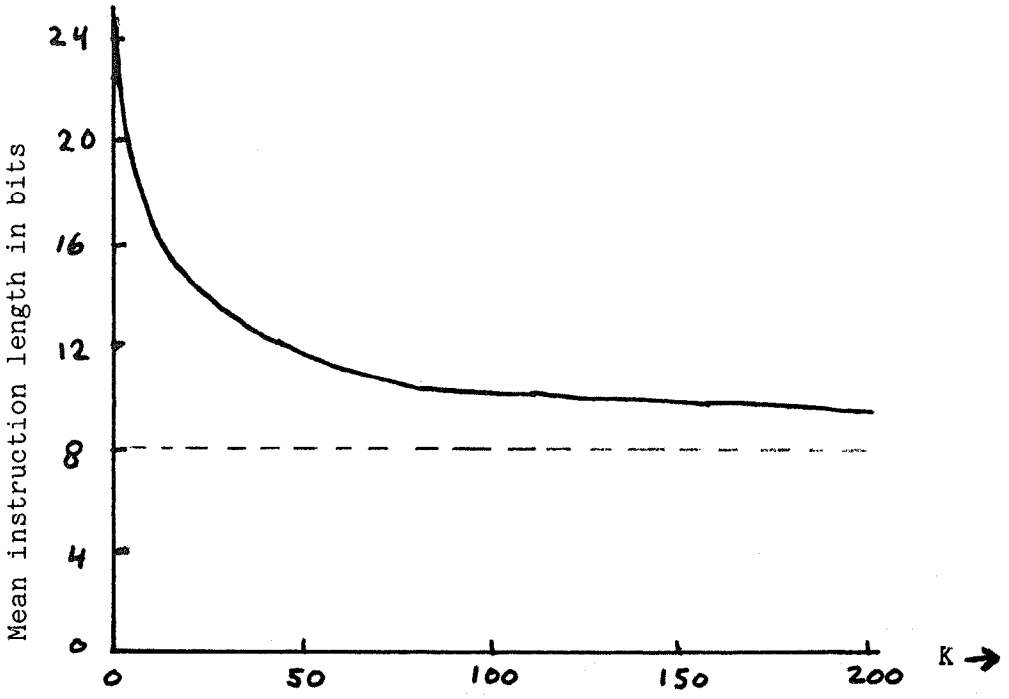

Fig. 1. Mean instruction length as a function of K. $M = 94$.