ON THE USE OF STACKS IN THE EVALUATION OF EXPRESSIONS

J. L. Keedy (Department of Computer Science, Monash University, Clayton, Victoria, 3168, Australia)

Part 1 - A Case for Stack-Oriented Instruction Sets.

Recently Myers presented a case against the use of stack-oriented instruction sets (and register-oriented instruction sets), favouring instead the two address store-to-store instruction format [1]. He argued that most assignment statements in most real programs involve zero or one arithmetic (or logical) operator, and therefore that 'textbook' examples of long expressions used to demonstrate the superiority of stack-oriented instruction sets are 'irrelevant'.

The view is open to several objections but here I raise only two of them. First, Myers's comparison included only one, not particularly favourable, version of stack instruction sets. Another version, in which the stack is combined with an accumulator, is superior to both the register and stack instruction sets considered by Myers. The accumulator/stack technique uses a single accumulator as an implicit operand for all arithmetic (and logical) operations. The second operand is either supplied explicitly with the instruction (e.g. as a literal constant or a memory address), or may be related implicitly to the stack top. With an explicit memory address operand instructions behave as on a normal single accumulator machine, so that LOAD A causes the value at the address A to be loaded into the accumulator, ADD A causes the value at the address A to be added into the accumulator, etc. This accumulator-oriented technique handles well expressions which do not require the saving of intermediate results. Stacks, on the other hand, are most useful exactly when such results have to be saved, and to handle these situations the accumulator can be coupled to a hardware-assisted stack. To achieve this each (appropriate) instruction code can be considered as having a special 'top-of-stack' (TOS) bit, which, when set, indicates a short instruction in which the normal memory operand is replaced by the value currently at the top of the stack; the result is left in the accumulator and the TOS operand is popped. Thus ADD(TOS) causes the value at the stack top to be added into the accumulator, and at the same time to be popped from the stack. When used in this way the accumulator acts as an additional stack cell on top of the actual top-of-stack, and to complete this mode of use we need a 'push and load' operator (which acts like a conventional push by pushing the accumulator value onto the actual stack top and then reloading it from the operand supplied) and a 'store and pop' operator (which acts like a conventional pop by storing the accumulator into the supplied address and reloading it from the stack top, which is popped). Apparently ambiguous function codes corresponding to PUSH&LOAD (TOS), STORE&POP (TOS), LOAD (TOS) and STORE (TOS) need not be wasted, since they can easily be microcoded to perform special stack top operations corresponding on traditional stack machines to functions such as: exchange top two stack locations, erase stack top, duplicate stack top, etc. (but in this case regarding the accumulator as the stack top).

Tables 1 to 3 reproduce Myers's comparisons of the 'typical' assignment statement encodings, with a fourth column added for the accumulator/stack instruction set. (Following Myers, function codes are 8 bits long, register addresses 4 bits and memory addresses 20 bits). Since it is arguable whether an extra bit should be added to the function code length of the accumulator/stack method to allow for special TOS

addressing, the bracketed figures of that column indicate the effect of this. On all Myers's criteria (instruction fetches, code compactness, and number of elements to be decoded) the accumulator/stack method performs as well as, and mostly improves upon, the normal stack and register methods. Comparison with the store-to-store method shows that the latter is superior in instruction fetches, and for the first two examples in the number of elements to be decoded. Regarding code compactness, the store-to-store method is slightly better for A:=B, and slightly worse for A:=B+C, but significantly better for the special (but admittedly frequent) case A:=A+B.

The store-to-store instruction format used by Myers, like the other techniques, assumes fixed length operands, but since he claims for this method some advantage of variable length operands, it is interesting to see the effect of adding a length field of say 8 bits to each store-to-store instruction, as is shown by the bracketed figures in the store-to-store column of the tables. Whilst this admittedly makes the instruction set more powerful than the others, in code compactness and element counts it more or less brings the accumulator/stack onto a par with the store-to-store format, with A:=B about equal, A:=A+B worse by 28 bits and 2 elements, and A:=B+C better by the same margin.

It seems on the evidence so far that although the accumulator/stack technique improves upon the stack and register techniques it is slightly inferior to the store-to-store technique (unless we make a somewhat doubtful use of length fields). But this brings me to the second objection, namely, that the evidence adduced by Myers is heavily biased in favour of the store-to-store technique. There is, for example, another extremely common operation performed in many programs, the interchange in the values of two variables (A:=:B). Unfortunately, widely used high level languages do not support an appropriate construct*, and so instead it gets coded as (TEMP:=A;A:=B;B:=TEMP) which produces for each interchange three zero operator assignments (and consequently falsely boosts statistics for simple assignments). Table 4 has been added to provide an encoding of an interchange, which shows that the accumulator/stack method performs better than all the other methods, especially the store-to-store method (except in instruction fetches). It also seems unreasonable to exclude from the comparisons at least one straightforward expression requiring an operand to be saved in a temporary location, because such expressions do undoubtedly appear occasionally in many programs, and frequently in some. Table 5 has therefore been added to illustrate A:=(B+C)*(C-D), and once again the accumulator/stack technique comes out ahead by a considerable margin, with the store-to-store technique trailing a poor fourth. We conclude that the accumulator/stack method of evaluating expressions is consistently better than the stack and register methods for all expressions analysed. It produces more compact code than the store-to-store method for A:=B+C, A:=:B and A:=(B+C)*(D-E), marginally less compact code for A:=B (except where this is a hidden interchange) and substantially less compact code for A:=A+B. It offers the further advantages that temporary storage is conveniently organised for any size and nesting of expressions, and that if combined with a procedural stack it can easily and naturally handle the evaluation of functions contained in expressions. In modern microcoded computer architectures, where

*This is a deficiency in language designs which should not affect comparisons of architectures (unless we are only concerned with building COBOL, FORTRAN, ALGOL or PL/1 machines in the future).

orthogonal instruction formats no longer provide significant advantages, it would seem feasible to provide a mixed instruction set which is based primarily on the accumulator/stack technique but which provides a more compact encoding of operations such as A:=A+B and A:=B (probably as store-to-store instructions*).

Part 2 - A Practical Example of the Accumulator/Stack (compared with a Pure Stack)

The most prominent current example of an accumulator/stack architecture is that found in the ICL2900 Series [2]. Rather curiously, this system only partly implements the previously described architecture, the main omission being the absence of a 'store and pop' operator; also, a stack top operand is not optimally encoded, but takes the short-form addressing field also used for accessing operands relative to the current local name base of the stack. On the other hand, the ICL2900 Series takes advantage of the technique by loosely coupling an index register and a descriptor register to the stack top in a manner analogous to the accumulator, the effect of which - if carefully used - is that the system appears to have separate stacks for computing values, indices and addresses.

In an earlier contribution Doran has implied that the ICL2900 computational technique is inferior to the pure stack technique of the B6700 [3]. Leaving aside certain unproven and highly dubious implications in that paper**, we find that the B6700 and ICL2900 instruction sequences for generating the five previously illustrated assignment statements are identical with those shown in cols. 1 and 4 respectively of Tables 1 to 5 (except that the MULT (TOS) operator is implemented on the ICL2900 as a normal MULT operator with TOS operand and STORE&POP is replaced by a simple STORE). Hence in terms of instruction fetches and number of elements to be decoded the ICL2900 is always superior to the B6700. The question of code compactness is complicated by the actual encoding of function codes and of address, which are quite different in the two machines. However, if we assume that all operands are reasonably accessible on the stack then each ICL2900 instruction will be encoded in 2 bytes (16 bits) and each B6700 instruction requiring an explicit operand will also require 2 bytes (16 bits) whilst B6700 instructions with implicit operands require 1 byte (8 bits)***. Under these assumptions

---

* Notice also that the most efficient encoding of an interchange is also likely to be a store-to-store SWAP A,B.

** For example, on issues related to this paper Doran implies (i) that the B6700 tagging mechanism is a greater boon to compiler writers than the ICL2900 accumulator size register (which he fails to mention); (ii) that the encoding of ICL2900 functions requires more bits than the B6700 (in fact the ICL2900 uses 7 bits, the B6700, 8 bits - though the situation is rather more complicated than this suggests); and (iii) that the two B6700 top-of-stack registers provide an advantage in reducing stack/memory references (whereas the equivalent ICL2900 systems provide a 256-word fast slave memory for the stack).

*** For the B6700 this gives access to variables at displacements upto between 512 words and 8 K from the display registers, depending on the current lexical level; for the ICL2900 it gives access to variables up to a displacement of 128 words from the local name base, and the stack top.

the length of a B6700 assignment sequence (ignoring indexing operations and the possibility of function calls) is:

$$(2a + f + 3) \text{ bytes} \qquad (1) \quad \text{B6700}$$

where a = the number of operands in the expression,

and f = the number of arithmetic (or logical) operators in the expression.

(The three extra bytes are required to bring the address of the left hand side to the stack top (2 bytes) and to store the result into that address (1 byte)). The equivalent length of an ICL2900 assignment sequence is:

$$(2a + 2i + 2) \text{ bytes} \qquad (2) \quad \text{ICL2900}$$

where a = the number of operands in the expression,

and i = the number of intermediate results which are not immediately reused (i.e. the number of 'stack and load' instructions).

(The two extra bytes are to store the result into the variable on the left hand side of the assignment).

Subtracting (2) from (1) we find that a B6700 sequence is longer than its ICL2900 equivalent by a difference d, where

$$d = f - 2i + 1 \qquad (3)$$

In those cases where intermediate results (if any) can always be immediately reused, this reduces to

$$d = f + 1 \qquad (4)$$

so that in this case a B6700 sequence is always at least one byte (where $f = 0$, i.e. A:=B) longer than its ICL2900 counterpart, and if f is large then the difference can be substantial. Where there are intermediate results which have to be saved, each such result implies an operator to generate it, and an operator to use it later; furthermore the fact that it cannot be immediately reused implies that at least one more operator must be present in the expression, hence

$$f \geq 2i + 1 \qquad (5)$$

Substituting (5) into (3) we find that where an expression contains intermediate results which are not immediately reused, the B6700 will always require

$$d \geq 2i + 1 - 2i + 1$$

$$\text{i.e. } d \geq 2 \qquad (6)$$

at least 2 bytes more than the ICL2900.

Hence under the addressing assumptions stated previously, a B6700 assignment statement will be encoded in at least one byte more than the ICL2900 (for the simple assignment case A:=B), and will usually be longer by between 2 bytes and (f+1) bytes, depending on the number of 'push and load' operators required on the ICL2900. It must be emphasized,

however, that for the vast majority of cases the difference will be minimal. Attempts to make a more general comparison, using the multiplicity of addressing modes available on the two systems, are bound to fail. Examples can be brought forward to favour either system. For example, in a highly block-structured program the B6700 encoding might turn out to be more compact because the ICL2900 can only support two lexical levels with ease, whereas the B6700 can support 30. On the other hand the ICL2900 would probably improve even more on the B6700 if operands are embedded in an array because its index and description registers can be coupled to the stack.

Conclusion. In Part 1 it was shown that the accumulator/stack computational technique is usually superior to normal stack or general register techniques, and, given a reasonable basis of comparison, that it compares favourably with the store-to-store technique except in the special case of A:=A+B, which can be overcome by providing a special instruction format to optimise this type of expression. In Part 2 we considered a particular example of an accumulator/stack machine, and found that despite its only partial implementation of the concept, and despite implications to the contrary, it performed better than a pure stack machine.

It is evident, however, that the manifest inferiority of the stack-type systems on such an important case as A:=A+B suggests that we have not yet found an ideal computational architecture for evaluating expressions. Equally, the interchange operation A:=:B shows up the inadequacy of most high level languages as a means of expressing satisfactory abstractions of common operators (another which comes to mind is the absence in high level languages of a construct which returns both a quotient and a remainder in integer division). These two areas might well provide fruitful objectives for future research.

Acknowledgement

Thanks are due to Professor C.S. Wallace for reading two drafts of this paper and especially for his helpful suggestions which have clarified several points.

References

1. G.J. Myers "The Case Against Stack-Oriented Instruction Sets", Computer Architecture News, Vol. 6, No. 3, August 1977.

2. J.L. Keedy "An Outline of the ICL2900 Series System Architecture", Australian Computer Journal, Vol. 9 , No. 2 , July 1977.

3. R.W. Doran "The ICL2900 Computer Architecture (compared with the B6700/7700)", Computer Architecture News, Vol. 4, No. 3, Sept. 1975.

	Stack	Register	Store-to-Store†	Accumulator/Stack⊗
	PUSHAD A	LOAD R1,B	MOVE A,B	*PUSH&LOAD B
	PUSH B	STORE R1,A		*STORE&POP A
	STORE			
Instructions	3	2	1	2
Size	64	64	48 (56)	56 (58)
Elements	5	6	3 (4)	4

Table 1. The Statement A:=B

	Stack	Register	Store-to-Store†	Accumulator/Stack⊗
	PUSHAD A	LOAD R1,A	ADD A,B	*PUSH&LOAD A
	PUSH A	ADD R1,B		ADD B
	PUSH B	STORE R1,A		*STORE&POP A
	ADD			
	STORE			
Instructions	5	3	1	3
Size	100	96	48 (56)	84 (87)
Elements	8	9	3 (4)	6

Table 2. The Statement A:=A+B

	Stack	Register	Store-to-Store†	Accumulator/Stack⊗
	PUSHAD A	LOAD R1,B	MOVE A,B	*PUSH&LOAD B
	PUSH B	ADD R1,C	ADD A,C	ADD C
	PUSH C	STORE R1,A		*STORE&POP A
	ADD			
	STORE			
Instructions	5	3	2	3
Size	100	96	96 (112)	84 (87)
Elements	8	9	6 (8)	6

Table 3. The Statement A:=B+C

	Stack	Register	Store-to-Store †	Accumulator/Stack ⊗
	PUSHAD A	LOAD R1,B	MOVE TEMP,A	*PUSH&LOAD A
	PUSH B	LOAD R2,A	MOVE A,B	PUSH&LOAD B
	PUSHAD B	STORE R1,A	MOVE B,TEMP	STORE&POP A
	PUSH A	STORE R2,B		*STORE&POP B
	STORE			
	STORE			
Instructions	6	4	3	4
Size	128	128	144 (168)	112 (116)
Elements	10	12	9 (12)	8

Table 4. The Statement A:=:B

	PUSHAD A	LOAD R1,B	MOVE A,B	*PUSH&LOAD B
	PUSH B	ADD R1,C	ADD A,C	ADD C
	PUSH C	LOAD R2,D	MOVE TEMP,D	PUSH&LOAD D
	ADD	SUBT R2,E	SUBT TEMP,E	SUBT E
	PUSH D	MULT R1,R2	MULT A,TEMP	MULT (TOS)
	PUSH E	STORE R1,A		*STORE&POP A
	SUBT			
	MULT			
	STORE			
Instructions	9	6	5	6
Size	172	176	240 (280)	148 (154)
Elements	14	18	15 (20)	11

Table 5. The Statement A:=(B+C)*(D-E)

---

* PUSH&LOAD and STORE&POP operators marked with an asterisk can be reduced to (presumably faster) LOAD and STORE operations respectively if the previous accumulator value is not needed again. This optimisation is not usually available on a pure stack machine, and on the other hand extra instructions are required on the register machine to save the previous contents of registers.

† Bracketed figures in the store-to-store column show the effect of adding an eight-bit length field to appropriate instructions.

⊗ Bracketed figures in the accumulator/stack column show the effect of treating the TOS bit as a ninth bit in the function code.