A STACK ADDRESSING SCHEME BASED ON WINDOWING

W. F. WONG

Department of Information Systems
and Computer Science
National University of Singapore
Kent Ridge, Singapore 0511.

1. INTRODUCTION

Discussions in [10]-[11], [5]-[7] has demonstrated that zero-operand stack instructions are, in general, inferior to one and two-operands' ones. Keedy [8] further proposed a stack-based instruction set which includes zero, one and two-operands. However, in the paper no concrete implementation was discussed.

We propose an implementation of a stack addressing scheme similar to the idea of register windowing [4] used in RISC. The underlying instruction set is similar to the Burroughs B6700 [12], involving the idea of displays. For the argument of its superiority over the original B6700's zero-operand addressing, we will fall back on [8]. Further, we shall argue that it is superior to the scheme proposed there, the typically RISC scheme and the memory-to-memory addressing scheme. We will also discuss some practical issues involved in the scheme's actual implementation.

2. THE PROPOSED ADDRESSING SCHEME

The proposed addressing scheme involves the maintenence and use of two registers which we will call the source and target window registers. This will be abbreviated to SWR and

TWR respectively. Associated with these registers are the register valid flags, SWR-F and TWR-F respectively. Their uses will be explained shortly. There are two new instructions that set these registers. The formats of these instructions are as follows :

SSW *ssssdddd dddddddd*

STW *ssssdddd dddddddd*

The opcodes (and all the opcodes discussed here) are assumed to be 8 bits in length. This is followed by a 2 byte operand. The highest 4-bits of these, *ssss*, is a display register number and the remaining 12 bits, *dddd dddddddd*, is a displacement into the stack section pointed to by *ssss*. (Note that B6700 addresses are 5 plus 9 bits instead of our 4 plus 12.) Specifically, the content of the display *ssss* is added to *dddd dddddddd* and the result is placed in the window register. By adhering to the B6700 addressing scheme in setting the window registers, we avoid improper access to stack sections invisible at the current lexical level. After establishing the window registers, we can now use the data within the window by attaching a 1 byte specifier to arithmetic and logical operators. This is done as follows :

opcode tcdddsss

where *opcode* is the original 8-bit dyadic opcode from the zero-operand instruction set. In addition, an extra bit is needed to distinguish windowing instructions from the original zero-operand ones. This makes the final length of *opcode* 9 bits. In the original B6700 scheme the two operands are taken from the top of the stack and the result replaces them. The specifier itself consists of 4 fields, a single bit *t* flag, a single bit *c* flag, a 3-bit *ddd* field and a 3-bit *sss* field. These fields are used as follows :

sss field - if *sss* = 7 and *c* flag is cleared, the first operand required is at the top of the stack. If *sss* is less than 7 and *c* flag is cleared, the first operand is fetched from the memory location given by [Content of SWR] + *sss*. If *c* is set, the first operand is the unsigned integer *sss*.

ddd field - if *ddd* = 7, the second operand is at the top of the stack, otherwise it is fetched from [Content of TWR] + *ddd*.

t flag - if it is clear , result of operation is placed at the top of the stack; if it is set result is moved to the memory location given by [Content of TWR] + *ddd*.

c flag - if it is set, *sss* if taken as a short unsigned integer literal; if is cleared then the first operand is fetched from the memory location given by [Content of SWR] + *sss*.

It can easily be seen that the above scheme covers the functionalities of zero, one and two operands. In addition, short literals are provided for. As an example, consider the assignment statement $A := (B + C) * (D - E)$. It will be compiled into the following sequence of pseudo-

instructions :

ADDW 00 <Displacement of B> <Displacement of C>

SUBW 00 <Displacement of C> <Displacement of D>

MUL

STOW 10 <Displacement of A> 111

Note that the total amount of space required is 60 bits plus the 48 bits needed to set the window registers giving a total of 108 bits, whereas that required by Keedy's scheme is 156 bits, assuming 24-bit addresses, 248 bits for straightforward MIPS code [2], 176 bits for VAX code [1], assuming word displacement and 104 bits in a stack addressing scheme similar to the original B6700 scheme. Another point must also be noted : the 6 bytes overhead in the set up of the window registers can be amortised over the amount of code that uses data within the windows, i.e. without resetting the registers.

3. AN ANALYSIS

Using the parameterised analysis found in [8], we can perform a similar analysis on the new scheme. In our analysis, we paramterised the opcode size while maintaining constant the specifier described in section 2 at 8 bits. First, some definitions :

t = the size of the opcode in bits.

s = the size of the operand address in bits. Here, $s = 8$.

f = number of instructions that address their operands within the same window.

The amortised cost of the window setting instructions per instruction that addresses operand within the windows is $(2(t+1) + 32)/f$ bits per instruction.

Table 1 can be used to compare with that found in [8]. The weights involved were proposed in [11].

The total weighted length as computed in Table 1 is

$$(1.283 + (2 / f)) t + (11.547 + (34 / f))$$

Keedy's calculation revealed that the weighted lengths of memory-to-memory instruction and his proposed instructions are :

$$1.310t + 2.620s$$

$$1.283t + 2.566 + 2.283s$$

respectively where s is the number of bits in the operand address. Let us now examine the length of the operand address necessary to produce the same or shorter equivalent code to the new scheme. Assuming a 8 bit opcode, i.e. $t = 8$, and at least two instructions uses data in the window,

i.e. $f = 2$, we have :

$s = 13.87$ for memory-to-memory instructions

$s = 14.88$ for Keedy's instructions

For today's applications, the above are insufficient. This is evidence of the brevity of the new scheme.

4. IMPLEMENTATION CONSIDERATIONS

One might object to the new scheme for several reasons. This include the fact the compactness of the new scheme involves complications in decoding and the new overhead introduced because of the need to save up the window registers during procedure calls and context switching.

The objection about the complexity involved in the decoding logic is a valid one. However, the original addressing logic of the B6700 is rather simple because most of the instructions are zero-operand ones. Further, there is already provisions for more complicated addressing involving direct and indirect stuff words. The addition of the new scheme in effect only needs new logic to handle the first two bits in the specifier.

Next, we turn to the objection of extra saves during context switching and procedure calls. Because the window registers are presumably set only once during procedure entry, a method can be worked out to save a little of the work involved in saving these registers. For this method to work we need the SWR-F and TWR-F flags. Upon procedure entry, a procedure will execute the two set window registers instructions. Note that the SWR-F and TWR-F flags would have been made invalid when the caller executes the call. To eliminate the saves at procedure calls and context switching, the instructions, besides setting the registers will also save the register values into predetermined locations in the current stack section. The SWR-F and TWR-F will be set as valid. At a procedure return, the flags are made invalid. The calling procedure will continue to execute as per normal. However, when an instruction uses the window addressing specifier, an interrupt will occur because the flags are invalid. The bottom of the current stack section is passed to this interrupt's handler so as to made the locations in which the window registers were saved addressable. We realise that this is a slight modification of the usual B6700 interrupt handling sequence [12]. However, this method avoids resetting the window registers if they were left untouched by the called procedure and reduces the number of saves necessary during procedure calls and context switches.

Another possible objection is to the rigidity of the scheme, since TWR can contain only results and cannot be read from. One possible to solution is to set SWR and TWR to be equal. But this would mean losing 7 locations. A better solution to the problem would be the addition of a

window register management instruction. Quite simply, this instruction, XWR, is a zero-operand instruction that swaps the contents of the two window registers. Thus we can have all 14 locations to work with.

We now turn to the issue of exploiting the new scheme to its maximum. To do so, we must make sure that the number of times the window registers are reset is kept to a minimum. This would imply that most of the data references should occur within the windows. However, this problem is no different from that of register allocations in register machines : the language compiler should employ clever allocation algorithms for local data [13]. An added advantage of the scheme over registers is that all locations in the windows have physical addresses. Problems associated with registers and the address-of operator [9] in many high level programming languages are absent.

The reader may have noted that our scheme improves on the original zero-operand B6700 instructions by reducing the number of instruction necessary to bring operands to the top of the stack for operations. In doing so it incurs an overhead of setting the window registers. However, this overhead is amortised over the number of instructions that reference operands within the window. Also an additional bit is necessary to distinguish the original zero-operand instructions from the new windowing ones. In actual implementation, it may be possible to squeeze these into currently reserved opcodes, keeping to the 8 bit opcode.

We now turn to the issue of cache utilisation and efficiency. Because of the structure of the stack machine, the amount of caching necessary to guarantee good performance is quite small, namely a portion of the top of the stack. In our scheme, it makes good sense to perform predictive caching. At the execution of the set window instruction, the portion of memory corresponding to the windows should be brought into the cache in anticipation of its usage. This avoids the cache miss processing at the first windowing instruction. Since the windows are quite small, only 7 words each, the amount of additional cache required is small. In fact, in a VLSI implementation [3] where the cache is on-chip, we see no advantage of registers over our scheme. In fact, one can say that the objective of our scheme is similar to copying a portion of the stack into registers. However, one might argue that with proper optimization, the B6700 too would locate most of its frequently referenced operands within close proximity of one another. But if this is the case, the differences in the addresses of most of the B6700 instructions would be quite small and it would not be economical to specify the full addresses. Our scheme takes advantage of this fact. Another point worth noting is that techniques of peephole optimization [13] can be applied to shorten codes written in the original B6700 instruction set.

5. CONCLUSION

We have presented a practical implementation of zero, one and two-operand stack addressing. Utilising ideas from RISC and original B6700 ideas, this scheme is demonstrated to be superior to many other addressing schemes in term of compactness. We have also discussed some of the implementation issues involved. The new addressing scheme has been incorporated into the design of SARC (Structured Array and Register Computer) [14].

The author would like to express his thanks to Professor C.K. Yuen of DISCS for his constructive criticisms and his careful reading of the draft.

Assignment Statement	Weight	Encoding	Weighted Length
A := B	72.1%	$t'+8+(2t'+32)/f$	$(0.721 + (1.442/f))t' + (5.768 + (23.072/f))$
A := A + B	14.4%	$t'+8+(2t'+32)/f$	$(0.144 + (0.288/f))t' + (1.152 + (4.608/f))$
A := B + C	6.1%	$2t'+16+(2t'+32)/f$	$(0.122 + (0.122/f))t' + (0.976 + (1.952/f))$
A := (B + C) * (D - E)	2.7%	$4t'+32+(2t'+32)/f$	$(0.108 + (0.054/f))t' + (0.864 + (0.864/f))$
A := B + C + D - E	4.7%	$4t'+32+(2t'+32)/f$	$(0.188 + (0.094/f))t' + (1.504 + (1.504/f))$

Note : $t' = t + 1$.

TABLE 1
PARAMETERIZED LENGTHS OF ASSIGNMENT STATEMENTS

REFERENCES

[1] *VAX-11 Architecture Handbook*, 1979, Digital Press.

[2] T. R. Gross et. al., "Measurement and Evaluation of the MIPS Architecture and Processor", *ACM Trans. on Comp. Sys.*, vol. 6, No. 3, Aug. 88, pp 229-257.

[3] J. L. Hennessy, "VLSI Processor Architecture", *IEEE Trans. on Comp.*, vol. C-33, no. 12, Dec. 84, pp 1221-1246.

[4] M. G. H. Katevenis, *Reduced Instruction Set Computer Architectures for VLSI*. 1986, MIT Press.

[5] J. L. Keedy, "On the use of stacks in the evaluation of expressions", *Comp. Arch. News*, vol. 6, no. 6, Feb 78, pp 22-28.

[6] J. L. Keedy, "On the evaluation of expression using accumulators, stacks and store-to-store instructions", *Comp. Arch. News*, vol. 7, no. 4, Dec 78, pp 24-27.

[7] J. L. Keedy, "More on the use of stacks in the evaluation of expressions", *Comp. Arch. News*, vol. 7, no. 8, Dec 79, pp 18-22.

[8] J. L. Keedy, "An Instruction Set for Evaluating Expressions", *IEEE Trans. on Comp.*, vol. C-35, No.5, May 83, pp 476-478.

[9] B. W. Lampson, "Fast Procedure Calls", *Proc. of 1st Int'l Conf. on Arch. Support for Prog. Lang. and Op. Sys., Comp. Arch. News*, vol. 10, no. 2, Apr 82, pp 66-76.

[10] G. J. Myers, "The case against stack-oriented instruction sets", *Comp. Arch. News*, vol. 6, no. 3, Aug 77, pp 7-10.

[11] G. J. Myers, "The evalution of expressions in a storage-to-storage architecture", *Comp. Arch. News*, vol. 6, no. 9, Jun 78, pp 20-23.

[12] E. I. Organick, *Computer System Organization : A Structured Approach.* 1973, Academic Press.

[13] A. S. Tanenbaum et. al., "Using Peephole on Intermediate Code", *ACM Trans. on Prog. Lang. and Sys.*, vol. 4, no. 1, Jan 82, pp 21-36.

[14] D. W. Wall, "Register Windows vs Register Allocation", *ACM SIGPLAN 88 Conf. on Prog. Lang. Design and Implem.*, Jun 88, pp 67-78.

[15] C. K. Yuen, *A Preliminary Design of the SARC : A Structured Array Computer with A Stack and Register Configuration.* DISCS Technical Report TRD3/87. Mar 87. National University of Singapore.