AN EXTENSIBLE STACK-ORIENTED ARCHITECTURE FOR A HIGH-LEVEL LANGUAGE MACHINE

by

Robert P. Cook and Insup Lee

Computer Sciences Technical Report #397

August 1980

AN EXTENSIBLE STACK-ORIENTED ARCHITECTURE FOR A HIGH-LEVEL LANGUAGE MACHINE

Robert P. Cook* and Insup Lee

Computer Sciences Department and
Mathematics Research Center
University of Wisconsin-Madison
Madison, Wisconsin 53706

*The author is partially supported by U. S. Army contract DAAG29-75-C-0024 and National Science Foundation grant MCS-7903947.

Abstract

MCODE is a high-level language, stack machine designed to support strongly-typed, Pascal-based languages with a variety of data types. The instruction set is constructed for efficiency and extensibility and is based on an examination of common programming language operations. The architecture provides programmed control over both operand type selection and address field widths. In addition, right operand addressing is included to improve the size characteristics of MCODE instructions over those of traditional stack machines. The design is compared for efficiency with the instruction sets of the EM-1, Digital Equipment PDP-11 and VAX-11/780.

CR Categories: 4.12, 4.22, 4.9, 6.21

Keywords and Phrases: stack machine, computer architecture, addressing modes.

1. Introduction

With the growing use of high-level languages for systems and applications programming, computer instruction set design has moved from bit selection of internal CPU data paths to instruction sets which are oriented to common high-level language operations. Tanenbaum[11] discusses a stack machine(EM-1) designed with this philosophy. The EM-1 is intended to directly execute the code produced by the SAL compiler. SAL is a typeless systems programming language similar to BCPL[10]. In this paper, we have extended the EM-1 to provide an instruction set for a Pascal-based, strongly-typed, systems programming language, Modula[13], which was designed by Wirth and implemented by Cook[6]. Our Modula machine code, MCODE, not only provides extensible type operations but also maintains the efficiency of the EM-1. The EM-1 was designed based on an analysis of 300 procedures comprising 10,000 lines of code. The MCODE improvements are based on our analysis[9] of 3,581 Pascal procedures and functions with over 120,000 lines of program text.

The next section gives a brief EM-1 description which is followed by a discussion of the MCODE improvements. Also, the instructions used for expressions and Modula statements are illustrated. Finally, some comparisons are drawn with respect to other current architectures, including the PDP-11[1] and VAX[2].

2. Background

Tanenbaum designed the EM-1[11] to optimize the most frequently occurring high-level operations in programs as analyzed by himself, Knuth[8], Alexander and Wortman[3], and Wortman[14]. The most effective innovations in the EM-1 are encoding references to the first 12 bytes of local procedure storage and 8 bytes of static storage as single opcodes, array element accessing, and "if" statement comparison and branching. The hypothesis is that smaller code sizes will enhance faster program execution by better utilizing the bandwidth of CPU data paths. In addition, as the machine gets closer to the source language, compilers can produce more efficient code and can eliminate space-consuming peephole optimization routines.

Another important aspect of the EM-1 design is the notion of giving the programmer code improvement tools which are machine independent. In Knuth's Fortran analysis[8], he strongly suggested that program execution histories be automatically generated for each job. With Tanenbaum's machine organization, the programmer need only declare the most frequently used variables first in textual order to effect a performance improvement.

3. Extensions

The first problem that we found in trying to use the EM-1 design was its lack of a variety of data types. Modula provides the user with character, Boolean, long and short integer, and floating point operations. When the EM-1 is extended to encompass these operations, the 255 opcode limit is quickly exceeded. Our solution was to introduce modes of computation. A mode sets the CPU's fetch and execute microprogram to adapt to a particular data type such as floating or integer. A collection of 8-bit opcodes is provided to set the CPU mode. Therefore, a single "+" opcode suffices for all addition operations on any data type. The setting of the mode can be thought of as the replacement of the microcode jump table for a subset of the opcodes.

The mode approach is based on our observation that expressions are usually comprised of operands of the same type; thus, we expect that the space occupied by any extra instructions needed to set the mode will be offset by the savings in opcode space. Modes also provide an expansion and contraction capability for machine families. For instance, all floating point operations could be eliminated to build microprocessors intended for traffic control or a decimal mode could be added for commercial applications. For many environments, the savings in microcode space could be significant.

Our second extension was to provide direct addressing for right operands. According to all of the analyses, expressions tend to be simple. Tanenbaum found, for instance, that 31% of all assignment statements had a single term for a right hand side. Consider the evaluation of "a+b" on a typical stack machine. We must "push a", "push b", "pop b and add", and "replace a with result". The alternative is to "push a", "add b", and "replace a with result". This sequence not only saves an instruction fetch but also the redundant push and pop of "b" plus the instruction space. These savings will be replicated for every term in any expression which can be evaluated from left to right.

Finally, we have extended Tanenbaum's single byte addressing modes, provided an option to shorten address fields, improved subscripting, record and pointer referencing, and introduced some additional high-level language oriented constructs. In the next section, we will discuss operand addressing.

4. Operand Addressing

The three MCODE instruction formats are illustrated below:

```
        FORMAT 1:
 FORM 2,3,3     Ø,opcode,local address

        FORMAT 2:
 FORM 8         opcode [operands]

        FORMAT 3:
 FORM 8,8       255,opcode [operands]
```

In MCODE, addressing is partitioned into references to either static or local procedure storage. The MCODE machine uses byte addressing and has an address space of 2**32 bytes. The instruction formats are designed so that the most frequently occurring operations require a minimum of instruction space.

A format 1 instruction can address the first 8 16-bit words of the current procedure's activation record. The impact of this convention can be seen by noting that our results indicate that 97% of all procedures have fewer than 4 formal parameters and 9Ø% of all procedures have fewer than 4 local variables. Tanenbaum's short address convention for static variables was eliminated since the size of the static address space is not known until load time. However, the number of parameters and local variables is known at compile time. In addition, our analysis shows that 53% of all variable references were to local variables or parameters. To test the effect of this idea, we changed all the local variables in the Modula compiler to C[7] "register" variables which decreased each instruction reference by 16 bits. The compiler's size decreased by 10% and its compile rate went up several hundred lines per minute.

The format 2 and 3 instructions can have their operands on the stack or can have a right operand specification. Operand addressing is optimized in a fashion similar to that provided by the B1700[12]. The AMODE instruction sets the address field width to 8, 16, or 32 bits for references to either static or local storage. Note that program counter relative addressing is not affected. More than 90% of all Modula programs can use an AMODE which selects 8-bit local and 16-bit static addresses.

As an example, the 8-bit AMODE setting would save 8 bits per operand reference over the 16-bit addresses used in the PDP-11. The AMODE setting has no effect on indirect addressing on the stack. The VAX implements 8-bit address fields but an 8-bit selector is also required for a total of 16 bits.

A natural concern, however, is keeping AMODE set correctly. Since Modula has no "go to", the AMODE bookkeeping is easily maintained on the parse stack. Also, the procedure call instructions automatically save and restore mode information. In addition, the linkage editor is responsible for checking address field overflow if too small an AMODE is being used. MCODE implements the following addressing forms:

A	operands on the stack
B	{static|local}x{direct|indirect}
C	local direct
D	indirect address on the stack
E	32-bit absolute address
F	constant(8, 16, 32 bits)

G	constant(Ø-15)
H	{subscript|element} x B
	subscript-((sp↑)-1)*Mode size + EA)
	element -((sp↑)+EA) Effective Addr.
I	local x {direct,indirect,indirect x {subscript,element}}
J	8-bit jump offset
K	16-bit jump offset

Forms B and H cover accesses to simple variables, pointers, one dimensional arrays, and record elements occurring in static and local storage, or as parameters. Subscript addressing assumes a lower bound of one which is the most common case. For direct addressing, different lower bounds can be subtracted from the address field to produce the correct subscript calculation. Forms F and G are used for immediate addressing while forms E, J and K are used for program counter relative jumps and absolute addressing. Forms I and C are used with the format 1(8 bit) instructions. Form I can be used to access local variables, "const" simple parameters, "var" simple parameters, and array and record parameters.

Tanenbaum[11] recommends that references to global procedure variables be implemented by a microcode search of the procedure call back-chains. The claim is that this method eliminates the overhead of maintaining a static display. Based on our experience with implementations of Algol[5] and Pascal[4], a single reference to a global variable uses more time than that needed to update the display. The following code sequence is typical.

procedure entry:

```
        CONTROLBLOCK[SAVE]=DISPLAY[NEST]
        DISPLAY[NEST]=PB
```

procedure exit:

```
        DISPLAY[NEST]=CONTROLBLOCK[SAVE]
```

The first eight locations in static storage are used for the DISPLAY. According to our study, 89% of all procedures were not nested; 9% were nested one level; and 2% were nested 2 or more levels. Out of the 3,581 procedures that we examined, ten procedures were nested to 4 levels and no procedures were nested more than four levels. Therefore, a maximum of eight nesting levels was considered sufficient. Next, we will examine the format of the one byte instructions.

5. Local Variable References

We followed Tanenbaum's design by allocating 64 opcodes to special addressing. As we discussed previously, the local variable address space was set at 8 16-bit words, or a 3-bit address field. This left 3 bits for opcodes. These 8 opcodes were partitioned as follows:

PUSH	Form I	(sp↓) = (EA)
POP	Form C	(EA) = (sp↑)
ADD	Form C	(sp) += (EA)
SUB	Form C	(sp) -= (EA)
CMPB=	Form C,K	if (sp↑)=(EA) then (pc) += SE(K)
CMPB<>	Form C,K	if (sp↑)<>(EA) then

(pc) += SE(K)

The PUSH instruction uses two opcodes for direct or indirect references to simple variables, and two opcodes for indirect, or "var", references to arrays and records. The number of addressing modes for POP was decreased to one in order to increase the number of opcodes. In addition, we found that variable loads occur in a 2.7/1 ratio over variable assignments which indicates that POP is used less frequently than PUSH. The last four opcodes were assigned based on our frequency of use information. Out of all operator occurrences, "+" was used 21% of the time, "-" was used 9%, "=" was used 20%, and "<>" was used 10% of the time. According to Tanenbaum, the dynamic frequency of these operators is even higher. In conditional expressions, we found that "=" made up 32% of all operators and that "<>" was used 17% of the time. Since Tanenbaum found that "if", "repeat", and "while" had a dynamic frequency of 38%, the comparisons were implemented to both test and jump. Using these formats, many subprograms can be completely coded using only 8 bit instructions.

6. Right Operand Addressing

Because of the number of opcodes needed for right-operand addressing, we restricted the operators based on the same frequency analysis which was used to select the 8-bit instruction set. The following table lists the instructions which can address memory:

PUSH	Form A,B,D,F,H,G	(sp↓) = (EA)
POP	Form A,B,D,H	(EA) = (sp↑)

PUSHA	Form B,E,H	(sp↓) = EA
ADD	Form A,B,F	(sp) = (sp)+(EA)
ADDTO	Form B	(EA) = (EA)+(sp↑)
AND	Form A,B,F	(sp) = (sp)&(EA)
CLR	Form B	(EA) = Ø
CMPB=	Form A,B,F	if (sp↑)=(EA)
CMPB<>	Form A,B,F	if (sp↑)<>(EA)
DEC	Form B	(EA) = (EA)-1
INC	Form B	(EA) = (EA)+1
MUL	Form A,B,F	(sp) = (sp)*(EA)
SUB	Form A,B,F	(sp) = (sp)-(EA)
SUBFM	Form B	(EA) = (EA)-(sp↑)

The selected operators make up 8Ø% of all operator references in the Pascal programs that we analyzed. Address modes B and F were chosen since 35% of all operand references were to simple variables and 34% of all operands were constants. The ADDTO and SUBFM instructions correspond to Modula statements.

7. Array, Record and Pointer References

Simple record references are treated just like simple variable references and can be accessed using direct addressing. However, arrays of records or records as parameters must be accessed by an offset from a base address. The "element" address mode implements the pointer or parameter case.

Our analysis showed that 16% of all array references had a single constant subscript and that 52% of all subscripts were a single simple variable. The constant subscript case resolves to a variable address so the standard address formats can be used to access the array. The "subscript" mode was introduced to implement accesses to one dimensional arrays. In fact, we found that references to multidimensional arrays made up only 7% of all array references. MCODE uses descriptors to implement the multidimensional case.

In the EM-1, every array has an array descriptor cell, an array descriptor packet and an array data area. This approach works fine for Algol but not for Pascal-like languages. First in Pascal, all arrays have static bounds so a single descriptor can be generated in static storage. This approach allows descriptors to be shared and saves stack space as well as setup time. Secondly, Pascal allows arrays of arrays and pointers to arrays which implies that the base address of an array may already be on the stack and not in a descriptor. The MCODE SUBS instruction transforms the subscripts into a single byte offset which can then be used by the PUSH or POP instructions. The SUBS instruction also checks each subscript for validity.

```
        SUBS            descriptor address
```

The instruction address points to an array descriptor which contains the number of bounds, bounds pairs, multipliers, element size and virtual origin. SUBS leaves the element index on the stack. For instance, "A[I].B[J]" would produce the following code.

```
        PUSH    I
        SUBS    A desc.
        PUSHA   element( A+B offset)
        PUSH    J
        SUBS    B desc.
        ADD
```

For most Modula programs, each array type can be described by a single instance of a descriptor no matter how many variables of that type are created. Next, the expression operators will be described.

8. Operators

The following table lists the MCODE operators which are all format 2 instructions.

ABSolute	LoGical Shift
ARith. Shift	MOD
CONVert	NEGate
DECrement	NOT
DIVide	OR
DUPlicate	SQuaRe
INCrement	XOR

MCODE also includes instructions for moving and comparing blocks of storage as well as library call instructions to implement the Modula virtual machine environment and the floating-point math routines. In the next section, the code generated for the "case", "if" and "for" statements will be discussed.

9. Statements

Procedure call and return are very similar to the EM-1, except for the display updating, and will not be described. The "if" statement is implemented with the following instructions:

CoMPare	= > < >= <= <>
CoMPare Branch	= > < >= <= <>
Branch	=Ø <>Ø
Branch	

As an example, the statement "<u>if</u> a<>b <u>then</u> inc(a) <u>end</u>" would generate the following code:

Instructions			Size	PDP-11		Size
PUSH	a		8	CMP	a,b	48
CMPB=	b	L1	24	JEQ	L1	16
INC	a		16	INC	a	32
			48			96

The syntax and code generated for the "for" statement are listed below.

```
for v:=e1 by e2 to e3 do S end
        PUSHA   v
        PUSH    e1
        PUSH    e2
        PUSH    e3
        FOR     L2
L1      S
        ENDFOR  L1
L2
```

The "case" instructions are as follows:

```
CASE            constant, offset
```

CASE	constant, constant, offset
CASETBL	constant, constant

These three forms cover the situations in which the "case" is distinguished by a single value, a range of values, or a jump table. Next, we will analyze the effectiveness of MCODE with respect to other machine designs.

10. Comparison with Other Machines

The results in Figure 1 extend the table in Tanenbaum[11] to include the VAX and MCODE. Obviously, the special addressing and descriptor-based array computations make a significant difference. MCODE performs better than the EM-1 for expressions and parameter referencing and is as good in all other areas. The difference in the "if" tests occurs because the EM-1 assumes a 2-bit field for branch offsets while we used an 8 bit field. The VAX instructions are computed using 8 bit displacement addressing. In addition, it should be pointed out that the VAX and MCODE are supporting many more data types than the PDP-11 or the EM-1. Figure 2 recomputes the space for the same statements but with all the machines forced to use 16 bit addressing.

The values in Figure 2 give a lower bound on the performance of MCODE whereas Figure 1 gives an upper bound on the difference. For 16-bit addressing, which would be used for references to static storage, MCODE is better in all categories. The EM-1 is forced to use a 16-bit opcode to access 16-bit addresses which results in its poor performance. Since 50% of all variable references are to static storage, we feel that this improvement could have a significant impact on execution speed. The VAX is still quite poor with respect to subscripting even though a special instruction is available for that purpose. Also, the figures do not reflect the dynamic effect of the savings since Tanenbaum's measurements indicate that the Figure 1 results are even more significant at runtime.

11. Conclusions

We feel that the availability of modes as an extension mechanism for high-level language machines can be a significant factor in adapting microprocessors to changing environments. Also, modes contribute to space efficiency in the instruction set. The use of address mode settings to reduce address field sizes and right operand addressing also contribute space savings. The current version of Modula produces PDP-11 or VAX code so we have the means to compare the exact statistics on the static and dynamic behavior of MCODE with these machines using the same programs in the same environment. Our analysis should contribute to the alternatives available for opcode design in modern machine families.

REFERENCES

[1] PDP-11 Processor Handbook. Digital Equipment Corporation, (1975).

[2] VAX11/780 Architecture Handbook. Digital Equipment Corporation, (1977).

[3] Alexander, W.G., and Wortman, D.B. Static and dynamic characteristics of XPL programs. Computer 8, (1975), 41-46.

[4] Burger, T.M. A portable optimizing Pascal compiler. M.S. Thesis, Vanderbilt University, (1978).

[5] Cook, R.P., Hansen, G. and Haynam, G. Extended ALGOL 60 reference manual. Vanderbilt University Computer Center Report, (1970).

[6] Cook, R.P. An introduction to modular programming for Pascal users. University of Wisconsin Technical Report 372, (Nov. 1979).

[7] Kernighan, B.W. and Ritchie, D.M. _The C Programming Language_. Prentice-Hall Inc., (1978).

[8] Knuth, D.E. An empirical study of FORTRAN programs. _Software--Practice and Experience 1_, 1(1971), 105-133.

[9] Lee, I. and Cook, R.P. Static Analysis of Pascal Programs. In preparation.

[10] Richards, M. BCPL: A tool for compiler writing and systems programming. _AFIPS SJCC V. 34_, AFIPS Press, Montvale, N.J., (1969), 557-566.

[11] Tanenbaum, A.S. Implications of structured programming for machine architecture. _Comm. ACM 21_, 3(March 1978), 237-246.

[12] Wilner, W.T. Design of the Burroughs 1700. _AFIPS FJCC V. 41_, AFIPS Press, Montvale, N.J., (1972), 489-497.

[13] Wirth, N. Modula: A language for modular multiprogramming. _Software--Practice and Experience 7_, 1(Jan. 1977), 3-35.

[14] Wortman, D.B. A study of language directed computer design. Technical Report CSRG-20, U. of Toronto, (1972).

Figure 1

Direct Addressing Instruction Size(in bits)

Statements	MCODE	EM-1	PDP-11	VAX
i:=Ø	16	8	32	24
i:=3	16	24	48	32
i:=j	16	16	48	4Ø
i:=i+1	16	8	32	24
i:=i+j	24	32	48	4Ø
i:=j+k	24	32	96	56
i:=j+1	24	24	8Ø	48
i:=a[j]	24	32	128	1Ø4
a[i]:=Ø	32	32	112	88
a[i]:=b[j]	4Ø	48	192	168
a[i]:=b[j]+c[k]	64	8Ø	3Ø4	248
a[i,j,k]:=Ø	64	48	176	2ØØ
if i=j then	32	24	64	64
if i=Ø then	24	16	48	48
if i=j+k then	4Ø	4Ø	112	96
if flag then	24	16	48	48
call p	24	16	64	32
call p(i) value	32	24	96	56
call p(i,j)	4Ø	32	128	8Ø
call p(i) byref	4Ø	32	112	56
for i:=1 by 1 to N do a[i]:=Ø end	1Ø4	88	176	116

Figure 2

16-Bit Address Fields

Statements	MCODE	EM-1	PDP-11	VAX
i:=Ø	24	32	32	32
i:=3	32	48	48	4Ø
i:=j	48	64	48	56
i:=i+1	24	32	32	32
i:=i+j	48	1Ø4	48	56
i:=j+k	72	1Ø4	96	8Ø
i:=j+1	56	8Ø	8Ø	64
i:=a[j]	72	96	128	128
a[i]:=Ø	64	72	112	1Ø4
a[i]:=b[j]	96	128	192	2ØØ
a[i]:=b[j]+c[k]	152	2ØØ	3Ø4	296
a[i,j,k]:=Ø	128	136	176	232
_if_ i=j _then_	64	96	96	8Ø
_if_ i=Ø _then_	48	64	8Ø	56
_if_ i=j+k _then_	88	136	16Ø	12Ø
_if_ flag _then_	48	64	8Ø	56