Computer Systems

G. Bell, S. H. Fuller, and D. Siewiorek, Editors

Implications of Structured Programming for Machine Architecture

Andrew S. Tanenbaum
Vrije Universiteit, The Netherlands

Based on an empirical study of more than 10,000 lines of program text written in a GOTO-less language, a machine architecture specifically designed for structured programs is proposed. Since assignment, CALL, RETURN, and IF statements together account for 93 percent of all executable statements, special care is given to ensure that these statements can be implemented efficiently. A highly compact instruction encoding scheme is presented, which can reduce program size by a factor of 3. Unlike a Huffman code, which utilizes variable length fields, this method uses only fixed length (1-byte) opcode and address fields. The most frequent instructions consist of a single 1-byte field. As a consequence, instruction decoding time is minimized, and the machine is efficient with respect to both space and time.

Key Words and Phrases: machine architecture, computer architecture, computer organization, instruction set design, program characteristics

CR Categories: 4.12, 4.22, 4.9, 6.21

Author's address: Computer Science Group, Vrije Universteit, Amsterdam, The Netherlands.

1. Introduction

Information about the way computers are actually used is of great importance to computer architects, programming language designers, and compiler writers. Whether or not a certain semantic primitive should be included in a machine's instruction set, made a language construct, or carefully optimized depends primarily upon its projected frequency of usage. This information can only be obtained empirically, since there is no way to predict a priori, whether, for example, REPEAT . . . UNTIL statements are more useful than CASE statements.

The ways in which certain programming languages are used has already been studied: Knuth [6] has examined Fortran; Salvadori, Gordon, and Capstick [9] have examined Cobol; Alexander and Wortman [1] have examined XPL; Wortman [15] has examined student PL.

In recent years unstructured programs have fallen into disrepute. A growing number of people have come to recognize the importance of structuring programs so that they can be easily understood. Although there is no generally accepted definition of structured programming yet (see [2] for discussion), most programmers intuitively realize that breaking programs up into small, easily understood procedures, and drastically reducing or even eliminating GOTO statements greatly improves readability. We are even beginning to see the development of new programming languages which have been intentionally designed without a GOTO statement [16].

In order to determine what characteristics structured programs have, it is necessary to collect and dissect a number of them. These data can then be used as a basis for designing computer architectures that can execute structured programs efficiently. The next section of this article describes a GOTO-less language we have developed to encourage good programming style. The third and fourth section contain an analysis of a collection of procedures written in this language. The fifth and sixth sections propose and discuss a machine architecture based upon our findings.

2. The Experiment

We have developed a typeless GOTO-less language (SAL) specifically intended for system programming [10]. It has been implemented [11] on a PDP-11/45, and used, among other things, to construct a general purpose time sharing system for that computer. The language resembles BCPL [8]; its control structures are similar to those of Pascal [5]. A summary of the executable statements follows.

Assignment
CALL
IF . . . THEN . . . ELSE . . . FI
RETURN

```
FOR . . . FROM . . . TO . . . BY . . . DO . . . OD
WHILE . . . DO . . . OD
REPEAT . . . UNTIL . . . LITNU
DO FOREVER . . . OD
EXITLOOP
CASE . . . IN . . . , . . . , . . . , OUT . . . ESAC
PRINT
```

Expressions are evaluated strictly left to right, with no precedence or parentheses. ELSE parts in IF statements are optional. RETURN statements exit the current procedure, and optionally return a value, so that a procedure may be used as a function. Procedures not returning an explicit value may terminate by "falling through", i.e. the END statement implies RETURN.

The WHILE statement tests at the top of the loop, whereas the REPEAT statement tests at the end of the loop. DO FOREVER statements are the same as WHILE TRUE DO; they are useful in operating system modules that endlessly get and carry out service requests, the "get" primitive blocking the process in the absence of a message. EXITLOOP is a forward jump out of one level of enclosing loop of any kind (FOR, WHILE, REPEAT, or DO FOREVER). Our experience indicates that this, plus RETURN, is sufficient most of the time. The CASE statement contains an integer expression that selects one of the clauses to be executed, or the OUT clause if the integer is out of range (as in Algol 68 [12]). There is no GOTO statement.

In addition to the above statements, there are a variety of declarations, debugging facilities and compiler directives.

The basic data types are machine words (including the general registers and the i/o device registers, accessible as the top 4K memory words), one-dimensional arrays of words and characters, bit fields, and programmer defined data structures consisting of a collection of named fields, each field being a word, character, bit field, or array. There are two scope levels, local (stack storage, reserved upon procedure entry, and released upon procedure exit), and global (static storage). A program consists of one or more procedures, and zero or more modules that declare and initialize external variables.

The programs examined for this research were all written by the faculty and graduate students of the Computer Science Group at the Vrije Universiteit. All the programmers involved made a very deliberate effort to produce "clean," well structured programs, knowing full well that succeeding generations of students would pore over their code line by line. This is clearly a different situation than one finds in the average, garden variety, computer center.

The amount of memory available on our PDP-11/45 was so small that the initial compiler could not handle procedures much larger than two pages of source code. This defect was remedied by declaring it to be a virtue, and by continually exhorting the programmers to produce short, well structured procedures. (The mean number of executable statements per procedure turned out to be 18.2). The combination of the GOTO-less language, the quality of the programmers, an environment with a long Algol tradition and no Fortran tradition, and our deliberate efforts to produce intelligible programs has resulted in what we believe to be state-of-the-art structured programs.

3. Characteristics of the Programs

For this study we have used a specially instrumented compiler to collect information on more than 300 procedures used in various system programs. Most of these were related to the time sharing system project. The results presented should be interpreted keeping in mind that operating system modules may systematically differ from say, applications programs, in certain ways, e.g. they have little i/o.

Where relevant, both static and dynamic measurements are given. Static measurements were obtained by having the compiler count the number of occurrences of the item in the source text. Dynamic measurements were obtained by having the compiler insert code into the object program to increment counters during program execution. The results are given in Tables I–VIII.

4. Discussion of the Results

According to our data, a typical procedure consists of 8 or 9 assignment statements, 4 calls to other procedures, 3 IF statements, 1 loop, and 1 escape (RETURN or EXITLOOP). Two of the assignment statements simply assign a constant to a scalar variable, one assigns one scalar variable to another, and 3 or 4 more involve only one operand on the right hand size. The entire procedure probably contains only 2 arithmetic operators. Two of the three conditions in the IF statements involve only a single relational operator, probably = or ≠.

The general conclusion that can be drawn from this data is the same as Knuth drew from his Fortran study: programs tend to be very simple. Combining this conclusion with the Bauer principle (If you do not use a feature, you should not have to pay for it), we suggest that most present day machine architectures could be considerably improved by catering more to the commonly occurring special cases. This will be discussed in detail in the next section. First we have a few more comments about the measurements.

In some cases there are significant differences between the static and dynamic measurements. Some of these differences are genuine, e.g. the operating system is constantly looking for internal inconsistencies in its tables. If an error is detected, an error handling

Table I. Percent Distribution of Executable Statements.

Statement Type	Static	Dynamic
Assignment	46.5	41.9
CALL	24.6	12.4
IF	17.2	36.0
RETURN	4.2	2.6
FOR	3.4	2.1
EXITLOOP	1.4	1.6
WHILE	1.1	1.5
REPEAT	0.5	0.1
DO FOREVER	0.5	0.8
CASE	0.3	1.2
PRINT	0.3	<0.05

Table II. Percent Distribution of Assignment Statement Types.

Type	Static	Dynamic
variable=constant	21.7	19.2
variable=variable	9.5	9.1
variable=function call	4.4	1.9
variable=array element	4.3	3.3
array element=constant	4.1	2.8
array element=variable	4.1	2.9
array element=array element	0.9	1.8
array element=function call	0.5	0.1
other forms with 1 rhs term	30.5	25.2
forms with 2 rhs terms	15.2	20.4
forms with 3 rhs terms	3.0	6.9
forms with 4 rhs terms	1.5	5.9
forms with ≥5 rhs terms	0.3	0.3

Table III. Percent Distribution of Operand Types

Type	Static	Dynamic
constant	40.0	32.8
simple variable	35.6	41.9
array element	9.3	9.2
field of structure	7.1	11.1
function call	4.8	1.6
bit field	3.2	3.3

Table IV. Percent Distribution of Arithmetic Operators.

Operator	Static	Dynamic
+	50.0	57.4
−	28.3	25.5
×	14.6	13.2
/	7.0	3.8

procedure is called. During normal operation there are no inconsistencies, so these error handlers are not called. These CALL statements increase the static number of CALL's but not the dynamic number.

Furthermore, an IF statement containing a single CALL statement in its THEN part and a single CALL statement in its ELSE part will be counted as one IF and two CALL's in the static statistics, but one IF and one CALL in the dynamic statistics, since only one branch is actually taken per execution. This effect increases the proportion of IF statements relative to other statements in the dynamic statistics.

On the other hand, a single loop executed 10,000 times gives grossly disproportionate weight to the statements in the loop in (only) the dynamic statistics. Thus the dynamic statistics may in fact be based on a very

Table V. Percent Distribution of Relational Operators.

Operator	Static	Dynamic
=	48.3	50.6
≠	22.1	18.6
>	11.8	10.2
<	9.5	9.0
≥	4.5	8.4
≤	3.8	3.3

Table VI. Percent of all Procedures with N Formal Parameters.

N	Static	Dynamic
0	41.0	21.2
1	19.0	27.6
2	15.0	23.3
3	9.3	10.8
4	7.3	8.8
5	5.3	6.6
6	2.3	0.6
7	0.3	0.2
8	0.3	<0.05
≥9	<0.05	1.0

Table VII. Percent of all Procedures with N Local Scalar Variables.

N	Static	Dynamic
0	21.5	30.7
1	17.2	26.5
2	19.8	15.4
3	13.5	4.2
4	8.3	4.9
5	5.3	10.0
6	4.6	1.6
7	3.6	1.0
8	1.3	1.6
9	1.0	0.8
10	0.7	<0.05
≥11	3.3	3.0

Table VIII. Percent Distribution of Number of Statements in "THEN" Part of IF Statements.

Statements	Static
1	47.4
2	20.5
3	9.9
4	5.8
5	2.3
6	3.4
7	1.2
8	1.1
9	2.0
≥10	6.1

much smaller sample than the more than 10,000 lines of source text used to derive the static statistics. For this reason the static statistics are probably more meaningful. In the remainder of this paper we will use the static statistics.

From the fact that 5.5 percent of the statements are loops, and 1.4 percent are EXITLOOP's, we estimate that at least 25 percent of the loops are "abnormally" terminated. (In addition, an unknown number of loops are terminated by RETURN). The

Table IX. Comparison of Static Executable Statement Distribution (percent).

Statement type	SAL	XPL	Fortran
Assignment	47	55	51
CALL	25	17	5
IF	17	17	10
Loops	6	5	9
RETURN	4	4	4
GOTO	0	1	9

Table X. Summary of EM-1 Instructions and Number of Opcodes Allocated to Each.

Instruction description	Format	1	2	3A
push constant onto stack		3	2	
push local onto stack		12	1	
push external onto stack		8	1	
pop local from stack		12	1	
pop external from stack		8	1	
zero address ADD, SUB, MUL, DIV		4		
increment local		12	1	
zero local		12	1	
increment top word on stack		1		
push array element onto stack			2	
pop array element from stack			2	
call			1	
load address			1	
load indirect			1	
mark		3	1	
advance stack pointer			1	
return		1		
for instruction				2
branch forward unconditionally		34	1	
branch backward unconditionally			1	
branch if operand 1 = operand 2		12	1	
branch if operand 1 ≠ operand 2		20	1	
branch if operand 1 ≤ operand 2		8	1	
branch if operand 1 ≥ operand 2		8	1	
branch if operand 1 < operand 2		4	1	
branch if operand 1 > operand 2		4	1	
branch if operand = 0		12	1	
branch if operand ≠ 0		20	1	
branch if operand ≤ 0		8	1	
branch if operand ≥ 0		8	1	
branch if operand < 0		4	1	
branch if operand > 0		4	1	
opcode 255 (i.e. use formats 3B, 4)		1		

discussion currently raging in the literature [7] about how premature loop termination should be incorporated into language syntax is not irrelevant.

Since measurements of the type presented in this paper are obviously very sensitive to idiosyncracies of one's programming style, it is interesting to compare our results to previously published work. Table IX compares executable statement distribution for 3 studies cited in Section 1. One difference between Fortran and the other languages stands out immediately: Fortran programs have relatively few procedure calls. This suggests that they are not well modularized. From Knuth's data (his Table I) we compute that the average Fortran subroutine has 86.3 executable statements, vs. 28.6 for XPL and 18.2 for SAL, which agrees with this hypothesis.

Our data gives an average of 0.45 arithmetic operators per expression, which agrees well with Alexander's and Wortman's figure of 0.41. Likewise, our measurement of 1.22 operators per conditional expression agrees with their value of 1.19 logical plus relational operators. Such good agreement enhances one's confidence in the universality of the results.

5. A Proposal for a Machine Architecture

Most present day computers have an architecture designed in the early 1960's. They have remained substantially unchanged for a decade in the name of compatibility in spite of their obstacles to generating efficient code from high level languages. A machine architecture based on the characteristics of the programs described in the previous sections is sketched below. The architecture is specifically intended for block structured languages that permit recursion, i.e. Algol-like languages.

Our architecture has two explicit goals: 1. minimizing program size, and 2. providing a target language to which compilation is straightforward. We choose to minimize program size rather than maximize execution speed for several reasons. First, execution speed depends not only on the raw clock rate, but also on the characteristics of the underlying microinstruction set. Given a high level language benchmark program and two proposed instruction sets, it is possible to determine unambiguously which object program is smaller, but not which is faster. (By hypothesizing a faster clock or better microarchitecture either machine can be speeded up). In other words, minimizing size is a more clearly defined goal than maximizing speed.

Second, size and speed are highly intertwined. All other factors being equal, a shorter program will execute faster than a longer one since fewer bits need be processed. If the memory bandwidth is N bits/sec and the mean instruction size is L bits, the maximum instruction execution rate will be N/L instructions/sec. The smaller L is, the faster the machine can be. Furthermore, on a machine with virtual memory, reducing program size reduces the number of page faults, which, in turn, reduces the time required to process the page faults, thereby speeding up execution.

Third, on large computers with sophisticated multiprogramming systems, a decrease in program size means an increase in the degree of multiprogramming, hence a higher CPU utilization, as well as less swapping.

Fourth, the small amount of memory available on minicomputers is often a serious limitation. Making the program fit into the memory may take precedence over all other considerations.

Fifth, on mini and micro computer systems, the cost of memory frequently is much larger than the CPU cost. Reducing memory requirements has a much

greater effect on total system cost than reducing execution time.

The fact that few compilers for third generation computers can produce code that even comes close to what a skilled assembly language programmer can generate argues strongly for redesigning machine architectures so that compilers can do their job better. (See [11] for some statistics). It is for this reason that we consider a stack machine, since generating efficient reverse Polish is simpler than generating efficient code for a register oriented machine. We assume the presence of a cache to eliminate the need for memory cycles when referencing the stack.

The design described below is intended for implementing modern programming languages such as Algol 60, Algol 68, Pascal, XPL, BCPL, SAL, and others of this genre, since they tend to facilitate rather than hinder the writing of well structured programs.

The proposed machine, which we shall call EM-1 (Experimental Machine-1) has a paged, segmented virtual memory. The program and data reside in different address spaces (like the PDP-11/45), so that instruction space segment 0 is distinct from data segment 0. An instruction space segment is a sequence of 8-bit bytes, each with a unique address. A data space segment is a sequence of words of N bits each (N is left unspecified here). The word length for data space segments may be different from that of instruction space segments. (See Table X.)

One data space segment is special: the stack. The stack has associated with it a stack pointer register (SP) that points to the top word on it. Whenever a procedure is entered, a new frame is allocated on the stack for the administration, actual parameters, and locals. The frame is released upon procedure exit. Figure 1 depicts the stack for the following Algol 60 program.

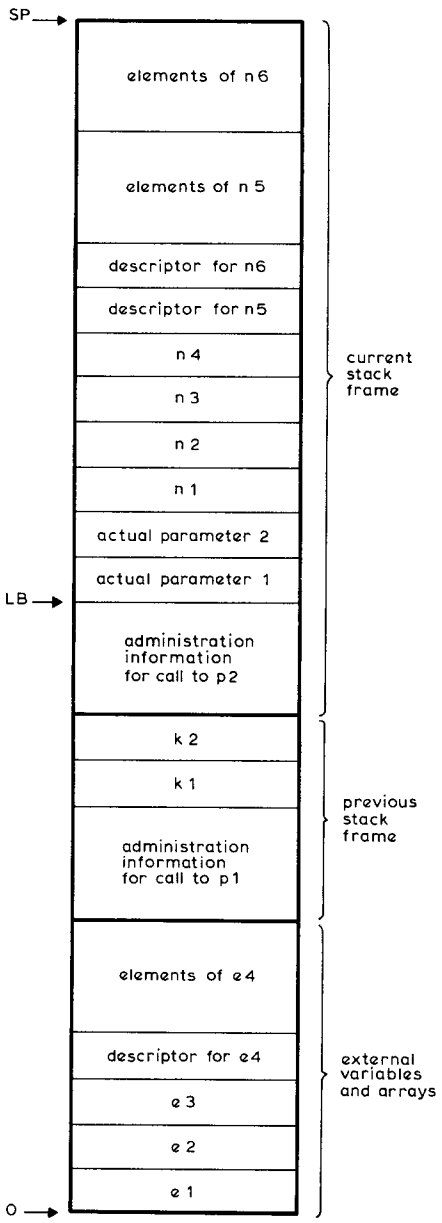
```
begin integer e1, e2, e3; integer array e4[1:3];
  proc p1;
  begin integer k1, k2; p2(k1, k2)
  end;
  proc p2(formal1, formal2);
  begin integer k1, k2; p2(k1, k2)
    integer array n5[1:4], n6[0:1];
    comment snapshot of Figure 1 taken here;
  end;
  p1
end
```

When $p2$ returns, SP will be reset to point to $k2$, thus removing that part of the stack marked "current stack frame" in Figure 1.

The stack frame for a procedure consists of 4 areas: (1) the administration information; (2) the actual parameters; (3) the local scalar variables and array descriptors; and (4) the elements of local arrays. The sizes of areas (1–3) are always known at compile time; the size of area (4) may not be known until run time.

A special hardware register, LB (Local Base) points

Fig. 1.



to the beginning of the local variables. Local variables are specified by giving their positions relative to LB.

The administration area contains the calling procedure's return address, the previous value of LB, and other (language dependent) information. It is assumed that the microprogram knows the size and organization of the administration area; a special instruction could be executed at the beginning of each program to tell it. Actual parameters can be addressed by giving their distance from LB, just as locals. Note that the administration area is not counted in order to reduce the size of the constants needed.

A procedure call takes place in the following steps:

1. A MARK instruction is executed to deposit the static and dynamic links on the stack. The MARK instruction has one operand which tells how much the static depth of nesting is increased or decreased.

This is needed to update the static chain. The MARK instruction also reserves space for the return address to be deposited subsequently.
2. The calling procedure pushes the actual parameters onto the stack.
3. A call instruction is executed, transferring control to the called procedure. The call instruction has as operand the index of a procedure descriptor, discussed later. This instruction must deposit the return address in the place reserved for it by the MARK instruction, update LB and transfer control.
4. The called procedure executes a single instruction that increments SP to reserve as much local storage as is initially needed; this instruction could also initialize the local variables to 0 or a special "undefined" value such as 1000 . . . 000 (two's complement −0). If more local storage is needed during execution of the procedure, e.g. for an Algol 68 local generator, SP can simply be advanced again.

We propose an addressing mechanism with distinct instructions for the 2 most important cases: local and external variables. Each instruction must provide an integer offset telling which variable is intended. Locals are offset above LB, and externals are offset from address 0 of the stack segment. For the purposes of addressing, procedure parameters are the same as locals.

Two other addressing forms are needed but are much less important. One is for full virtual addresses consisting of a segment and word within the segment. The other is for accessing intermediate lexicographical levels in block structured languages by means of a (relative lexicographical level, offset) pair. Rather than using a display, which must be frequently updated at considerable cost, we propose that at some position within the administration area known to the microprogram is the LB value of the most recent incarnation of the procedure in which the current procedure is nested (i.e. the static link). Given a (relative lexicographical level, offset) pair, the microprogram can follow the chain and locate variables at any outer static level. Note that the penalty for accessing intermediate levels is only a few microinstructions and one memory reference for each level of nesting followed. The combination of infrequent usage and a small penalty per use makes this method attractive since it reduces procedure call overhead, which is far more crucial.

The stack is also used for all arithmetic and logical operations, shifting, etc. An assignment is performed by first pushing the value to be assigned onto the stack (or perhaps its descriptor, if provision is made for assigning entire arrays in one instruction), and then popping it to its destination, a total of 2 instructions. The statement $A = B \times C$ is handled by 4 instructions: PUSH B; PUSH C; MULTIPLY; POP A.

The advantage of a stack type architecture for arithmetic is clear: compilers can translate expressions to reverse Polish very simply, with no complicated register optimization needed. High execution speed can be attained by a hardware cache memory that retains the most recently referenced words (i.e. the top of the stack) in high speed storage, or by having the microprogram keep them in its scratchpad memory. If the arithmetic expressions evaluated are simple, little cache or scratchpad storage will be needed. Our data indicate that 80 percent of all expressions consist of a single term, 95 percent consists of 1 or 2 terms and 99.7 percent consists of 4 or fewer terms, meaning that rarely will more than 4 operands be on the stack simultaneously.

Most of the instructions require an opcode and a small constant, which we call the "offset." The offset is generally used to select one of the local variables, one of the external variables, the number of bytes to skip (branch instructions), etc. The following five instruction formats are used by EM-1.

Format	Bytes	Description
1	1	byte 1 = opcode + offset (arithmetic sum)
2	2	byte 1 = opcode, byte 2 = offset
3A	3	byte 1 = opcode, bytes 2,3 = offset
3B	3	byte 1 = 255, byte 2 = opcode, byte 3 = offset
4	4	byte 1 = 255, byte 2 = opcode, bytes 3,4 = offset

The choice of machine instructions, and their assignment to formats, should be carefully arranged to minimize program size (based on the data of Section 3). In particular, an effort should be made to insure that the most common statements can be translated into 1 byte instructions most of the time. The scheme described below is constrained by the fact that the total number of format 1 instructions plus format 2 instructions plus format 3A instructions must not exceed 255. Some instructions, may occur several times in the order code, e.g. push constant onto the stack occurs in formats 1, 2, and 4, with a different range of constants provided in each form.

The idea of using shorter bit patterns for common instructions and longer bit patterns for infrequent instructions is not new. Huffman [4] gives a method for encoding items whose probabilities of occurrence are known, in the minimum number of bits. An approximation of this technique has been used in the design of the Burroughs B1700 S-machines (Wilner, [13, 14]). In the SDL S-machine, opcodes can be 4, 6, or 10 bits, and addresses 8, 11, 13, or 16 bits. A single address instruction can have a length of 12, 14, 15, 17, 18, 19, 20, 21, 22, 23, or 26 bits. Since the B1700 microarchitecture is extremely flexible (among other things being able to read an arbitrary length bit string—up to 24 bits—out of memory beginning at an arbitrary bit, in a single microinstruction) the use of peculiar length instructions does not slow down interpretation.

However, nearly all other computers are based upon a memory organization using fixed length words. For a microprogram with internal registers, bus widths etc. of 8, 16, or 32 bits interpreting a "machine"

language whose instructions came in units of 12, 14, 15, 17, 18, 19, 20, 21, 22, 23, or 26 bits would be unbearably slow, since nearly every instruction would straddle word or byte boundaries, necessitating time consuming shifting and masking operations to extract the opcode and address fields. The scheme described by Wilner is only feasible if every single bit in memory has a unique address, a situation which is rarely the case.

The instruction set of EM-1, in contrast, also provides a very efficient method for encoding instructions, but is based on a memory in which every 8-bit byte has a unique address, rather than every bit having a unique address. This makes the principles of the EM-1 design applicable to a much larger number of computers than one utilizing arbitrary length bit fields.

From Table I we see that the assignment, IF, CALL, RETURN and FOR statements together account for 96 percent of the source statements. Therefore we will design an instruction set to handle the object code from these statements efficiently. To push local variables (including parameters) onto the stack, we propose 12 distinct 1-byte (format 1) opcodes, one each for offsets 0–11. Twelve instructions allow access to all the locals (and parameters) in 94.6 percent of the procedures, and to more than 50 percent of the locals in the remaining procedures. For example, opcodes 114–125 might be used for PUSH LOCAL 0, PUSH LOCAL 1, . . . , PUSH LOCAL 11. There is no need to have distinct "opcode" and "address" bits.

Eight opcodes will be allocated to stacking the 8 external variables at the base of the stack segment. Since 81.4 percent of the constants in our data were either 0, 1, or 2, we allocate 3 opcodes for pushing these constants onto the stack.

At this point 23 of the 255 available 1 byte instructions have been used. Another 20 are needed for popping values from the stack. To handle programs with up to 256 locals, or externals, 4 format 2 instructions are needed: 2 push and 2 pop. Two more opcodes (format 2) are needed to push positive and negative constants up to 256 onto the stack. Format 4 (16 bit offset) can contain instructions with larger offsets for truly pathological programs. By including zero address (stack) instructions for add, subtract, multiply, and divide, we have sufficient instructions to evaluate most scalar expressions, using 53 of the opcodes.

Setting local variables to zero, and incrementing them by 1, are so common that we allocate 24 format 1 and two format 2 opcodes for this purpose. Incrementing the top of the stack is also worth an opcode.

Array accesses are accomplished using descriptors on the stack. Each descriptor (which may be 1 or more words, depending on N, the word length) contains the bounds and strides, S_i, for the array. For example, the address of $A[i, j, k]$ can be found from

$$\text{address} = S_0 + S_1 \times i + S_2 \times j + S_3 \times k$$

where the strides can be computed once and for all as soon as the bounds are known, at compile time in many cases, and at run time in the others. The descriptor must also contain the number of dimensions and the element size (and the segment number, for nonlocal arrays).

Array elements are accessed as follows. First the subscripts are stacked, requiring at least one instruction per subscript. Then a PUSH ELEMENT instruction is executed, specifying the offset of the descriptor from LB. This instruction removes all the subscripts from the stack, and replaces them with the selected element. The instruction also performs all bounds checking (unless disabled) and traps upon detecting a subscript error. A second opcode is needed for a POP ELEMENT instruction that first pops the subscripts and then the value. With these two instructions, the statement $A[I] := B[J]$ can usually be compiled into only 6 bytes of object code, including all bounds checking (PUSH J; PUSH ELEM; PUSH I; POP ELEM). This is a substantial improvement over most conventional designs. Four format 2 instructions are needed for pushing and popping local and external array elements.

Note that this addressing scheme is not affected by the size of the arrays. Assuming that a descriptor can fit in a single machine word, a procedure with ≤ 256 large arrays could nevertheless perform all array accesses using exclusively format 2 instructions.

For calling procedures, we envision one format 2 instruction whose offset is an index into a table held in a special data segment. Each table entry could contain the segment and address of the object code, possibly a "not yet linked" bit, to implement dynamic linking as in MULTICS, and possibly some protection machinery to keep less privileged procedures from calling more privileged ones. The symbolic name might also be present for debugging purposes and a counter to be incremented by the microprogram upon each call might be provided for performance monitoring.

To allow the instruction to locate the administration area in order to deposit the return address there, and to update LB, the number of words of parameters is also needed. For programs with up to 256 procedures, the call instruction will be 2 bytes, although a method to reduce this to 1 byte in most cases will be described below.

No additional instructions are needed for call-by-value. For call-by-reference an additional format 2 instruction to push an address onto the stack would be useful, along with one to fetch a parameter passed by reference (i.e. load indirect). The three most common types of procedure calls are to increase the depth of static nesting by 1, leave it unchanged, and decrease it by 1. Three opcodes are devoted to the three corresponding MARK instructions.

After a MARK instruction the distribution of the next few instructions is radically more different than

the normal one. This fact can be exploited to reduce the procedure call instruction to 1 byte in many cases, using a generalization of the idea of Foster and Gonter [3]. The only instructions than can follow a MARK instruction are those needed to pass the parameters, if any, and the CALL itself. Most parameters are constants, variables, or simple expressions, which can usually be passed using only a limited number of different instructions, mostly load type instructions. About 200 opcodes could be reserved for CALL's, each corresponding to a specific procedure descriptor. These CALL instructions would each require only 1 byte.

The simplest way to implement this would be to have the microprogram maintain the microaddress of the start of the instruction fetch loop in one of its registers. At the end of the execution phase of each interpreted instruction the microprogram would jump indirectly to this register. The MARK instruction would reload this register with the address of an alternative fetch loop, which would merely use a different branch table, in effect temporarily remapping the opcodes. The CALL instruction could restore the normal opcodes by resetting just one internal register. The use of opcode remapping can also be used in any other context with explicit first and last instructions.

An instruction with a 1-byte offset is needed by the called program to advance SP. The return instruction, which needs no offset, restores the stacked program counter and previous LB value (which are at known positions below the current LB) and resets SP.

Our proposed FOR statement instructions are based upon our measurement that 95 percent of the loops have a BY part of +1 or −1. Before the loop, the controlled variable is initialized, and the TO part is evaluated and pushed onto the stack. The EM-1 FOR instruction reads the TO part and the controlled variable. If the termination condition is met, a forward branch out of the loop occurs. Otherwise the controlled variable is updated and the next instruction is executed. The TO part is only removed from the stack when the loop is terminated. To allow both tests for both upward and downward counting, two opcodes are needed. (For languages in which the TO and BY parts may change during execution of the loop, variants of these instructions will be needed). Both instructions use format 3A. The offset of the controlled variable is in the second byte of the instruction, and the forward branch distance is specified in the third byte. The body of the loop is terminated by an unconditional branch backward to the FOR instruction.

At this point we must devise instructions to handle IF statements. A number of third generation machines perform conditional branching by first setting condition code bits, and then testing them in a subsequent instruction. EM-1, in contrast, combines these functions, and eliminates the need for condition codes.

There are three types of branch instructions, distinguished by the number of operands they remove from the stack. The unconditional branch forward and backward instructions do not remove any operands from the stack. The second group removes one operand and compares it to zero, branching forward if the condition specified by the opcode ($=, \neq, <, >, \leq$, or $\geq$) is met. This group is useful for statements such as IF N = 0 THEN . . . If Boolean variables represent FALSE by 0 and TRUE by 1, this group can also be used for statements such as IF FLAG THEN. . . .

The third group of branch instructions removes two operands from the stack, compares them, and branches forward if the specified condition is met. Backward conditional branches are not needed for translating IF statements (or WHILE statements either).

Each branch instruction specifies an offset which is the branch distance in bytes relative to the instruction itself. (Offset = k means skip $k + 1$ bytes.) Intersegment branches are prohibited, so that the procedure call mechanism can be used to limit access to privileged procedures. The size of the offsets required can be estimated from the data of Table VIII. Based upon the design proposed above, we estimate that the average source statement will require not more than 4 bytes of object code. This means that an offset with a range of 0–3 (i.e. 4 instructions) is sufficient for nearly half the IF statements, and a range of 0–15 (i.e. 16 instructions) is sufficient for more than $^4/_5$ of the cases. We need 14 opcodes to provide format 2 instructions for the unconditional branch, 1 operand conditional branch, and 2 operand conditional branch instructions.

This leaves 141 opcodes over for the format 1 opcodes. A possible allocation covering most of the frequently occurring cases is given in the summary of opcode usage below. If the average statement needs 4 bytes of object code, the division proposed below will handle 77 percent of the IF tests in a single byte. Note that "IF $A = B$" compiles into a branch NOT equal instruction to skip over the THEN part.

We will not discuss the instruction set further here. Suffice to say that all the instructions that could not be included in format 1 or format 2 for lack of encoding room, are included in format 3B. Also versions of all the above instructions should be provided as format 4 instructions (16-bit offset). Instructions needed, but not discussed above, e.g. accessing intermediate lexicographical levels of block structured languages should also be provided as format 3B and 4 instructions. There should also be instructions for multiple precision arithmetic, floating point, shifting, rotating, Boolean operations, etc.

It should be obvious that our design is not optimal in the information theory sense. More data and detailed simulation are needed to fine tune the choice of format 1 opcodes. On a user microprogrammable computer, one can envision tuning the format 1 instruction set to match the measured characteristics of impor-

tant production programs, and loading a special highly optimized microprogram before beginning program execution. Alternately, a whole collection of single chip microprocessors could be kept in house, each with a read only microprogram tuned to a different application.

6. Discussion of the Machine Architecture

Our major point in this whole discussion is to illustrate that 1 byte instructions in this design can often do the work of 4 byte or longer instructions in conventional machines. To illustrate the savings of EM-1, Table XI gives some examples of the size of the EM-1 code compared to DEC PDP-11 code and CDC Cyber code, as examples of mini and mainframe computers. The PDP-11 and Cyber code sequences used for comparison are those a good compiler might reasonably expect to generate in order to minimize object program size. It is assumed that these are fragments from a block structured language that permits recursion and requires subscript checking. All local variables are assumed to be on the stack, not in registers (except loop indices) and EM-1 is assumed to be able to use the shortest instruction format. Both the PDP-11 and Cyber make use of calls to run-time subroutines whose size is not counted here.

As a second test, 4 programs were carefully coded in assembly language for EM-1, the PDP-11 and the Cyber. In contrast to the above examples, these were complete programs, and the ground rules permitted the use of registers. There was no run time system (i.e. everything was coded in-line) and subscripts were not checked. The results are given in Table XII. It should be noted that the PDP-11 and Cyber test programs were carefully hand coded by an experienced assembly language programmer. Few compilers could ever generate object code this compact, whereas it would be easy to have a compiler generate the EM-1 code used in the examples due to the close match between the EM-1 instruction set and reverse Polish. This means that EM-1 is actually much better than the above data might at first indicate.

It is important to realize that in an environment consisting of many short procedures, the register sets provided by a third generation machine are of little value. They can be used for temporary results during expression evaluation, but from our data, that of Alexander and Wortman, and also Knuth's, one register is usually enough. The registers cannot be used effectively to hold local variables, because they must be constantly saved and restored upon procedure calls. This save-restore overhead will be very severe if, as our data shows, one out of every four statements is a procedure call.

Although we have not emphasized execution speed, a microprogrammed EM-1 machine is potentially very fast. The microprogram would fetch the opcode and then execute a 256-way branch. Since each of the format 1 instructions is relatively simple, each instruction could be handled by a small number of microinstructions. In contrast microprograms for machines like the PDP-11 and IBM 370 must do considerable extraction and manipulation of short fields within the target instruction. This is avoided in EM-1. By having a distinct microroutine for each of the twelve instructions that push a local variable onto the stack, none of these microroutines would have to do any decoding or bit extraction, providing for very fast execution. The other format 1 instructions would also be fast for the same reason. Alternately, to reduce the size of the microprogram at the expense of execution speed, all the target instructions of a given type could share one microroutine.

Table XI. A Comparison of EM-1, PDP-11, and Cyber Object Code Size (in Bits).

				Ratios	
Statements	EM-1	PDP-11	Cyber	PDP-11/ EM-1	Cyber/ EM-1
I := 0	8	32	45	4.0	5.6
I := 3	16	48	60	3.0	3.8
I := J	16	48	75	3.0	4.7
I := I + 1	8	16	60	2.0	7.5
I := I + J	32	48	90	1.5	2.8
I := J + K	32	96	105	3.0	3.3
I := J + 1	24	80	75	3.3	3.1
I := A[J]	32	128	120	4.0	3.8
A[I] := 0	32	112	105	3.5	3.3
A[I] := B[J]	48	192	180	4.0	3.8
A[I] := B[J] + C[K]	80	304	285	3.8	3.6
A[I, J, K] := 0	48	176	165	3.7	3.4
IF I = J THEN . . .	24	64	105	2.7	4.4
IF I = 0 THEN . . .	16	48	60	3.0	3.8
IF I = J + K THEN . . .	40	112	150	2.8	3.8
IF FLAG THEN . . .	16	48	60	3.0	3.8
CALL P	16	64	60	4.0	3.8
CALL P1(I) (by value)	24	96	90	4.0	3.8
CALL P2(I, J) (by value)	32	128	120	4.0	3.8
CALL P3(I) (by reference)	32	112	90	3.5	2.8
FOR I FROM 1 TO N DO A [I] := 0 OD	88	176	225	2.0	2.6

At first it may appear that producing code for EM-1 would give compiler writers nightmares, due to the multiple instruction formats. This problem can be easily solved by first writing an optimizing assembler that has a single mnemonic for "load local variable onto the stack" (*e.g.* LODLOC SYM), etc. The assembler, and not the compilers, chooses the shortest feasible instruction format. The assembler should also recognize sequences such as PUSH 0; POP X and PUSH X; PUSH 1; ADD; POP X and replace them by ZERO X and INCR X respectively. Compilers might also leave the task of sorting the local variables on number of occurrences, and assigning the most heavily used ones lower offsets to the assembler. Once such an assembler was written, it could be used as the last

Table XII. A Comparison of EM-1, PDP-11 and Cyber Object Code Size (in Bits)

Program	EM/1	PDP-11	Cyber	Ratios PDP-11/ EM-1	Ratios Cyber/ EM-1
Towers of Hanoi	352	992	2205	2.8	6.3
sort integer array	562	1248	1260	2.2	2.2
dot product	552	832	1140	1.5	2.0
find primes	306	704	1020	2.3	3.3

pass of all compilers, allowing them to produce straightforward reverse Polish, and still get locally optimal code.

7. Summary

There is a certain analogy between a Huffman code used to encode text in a minimal number of bits, and our proposal for a machine language with a compact instruction set. In both cases it is necessary to determine the frequencies of occurrence of the data to be encoded (letters and instructions, respectively) by empirical measurements. We have done this and reported the results in Section 3. Then an encoding scheme must be devised in which the most commonly occurring cases are assigned the shortest bit patterns, and the least commonly occurring cases are assigned the longest bit patterns. This is in contrast to a scheme in which all cases are assigned the same length bit pattern. In EM-1 the most frequently occurring instructions are encoded in a single byte, which is both efficient in storage and avoids the problems associated with variable length bit strings produced by true Huffman coding. This leads to object programs that require little memory and are capable of being executed very easily (i.e. fast).

Received February 1976; revised January 1977

References

1. Alexander, W.G., and Wortman, D.B. Static and dynamic characteristics of XPL programs. *Computer 8* (1975), 41-46.
2. Denning, P.J. Is it not time to define 'structured programming'? *Operating Syst. Rev. 8* (Jan. 1974), 6-7.
3. Foster, C.C., and Gonter, R.H. Conditional interpretation of operation codes. *IEEE Trans. Comptrs. C-20*, 1 (1971), 108-111.
4. Huffman, D. A method for the construction of minimum redundancy codes. *Proc. IRE 40* (1952), 1098-1101.
5. Jensen, K., and Wirth, N. *PASCAL User Manual and Report.* Springer-Verlag, New York, 1974.
6. Knuth, D.E. An empirical study of FORTRAN programs. *Software – Practice and Experience 1* (1971), 105-133.
7. Knuth, D.E. Structured programming with go to statements, *Computing Surveys 6* (1974), 261-301.
8. Richards, M. BCPL: A tool for compiler writing and system programming. Proc. AFIPS SJCC, Vol. 34, AFIPS Press, Montvale, N.J., 1969, pp. 557-566.
9. Salvadori, A., Gordon, J., and Capstick, C. Static profile of COBOL programs. Sigplan Notices (ACM) 10 (1975), 20-33.
10. Tanenbaum, A.S. A programming language for writing operating systems. Rep. IR-3, Wiskundig Seminarium, Vrije U., Amsterdam, 1974.
11. Tanenbaum, A.S. A general purpose macro processor as a poor man's compiler. *IEEE Trans. Software Eng. SE-2* (1976), 121-125.
12. van Wijngaarden, A., Mailloux, B., Peck, J.E.L., and Koster, C.H.A. Report on the algorithmic language ALGOL 68, Num. Math. *14* (1969), 79-218.
13. Wilner, W.T. Design of the Burroughs B1700. Proc. AFIPS FJCC, Vol. 41, 497, AFIPS Press, Montvale, N.J., 1972, pp. 489-497.
14. Wilner, W.T. Burroughs B1700 Memory Utilization. Proc. AFIPS FJCC, Vol. 41, AFIPS Press, Montvale, N.J., 1972, 579-586.
15. Wortman, D.B. A study of language directed computer design. CSRG-20, U. of Toronto, Toronto, Ont. (1972).
16. Wulf, W.A., Russell, D.B., and Habermann, A.N. BLISS: A language for systems programming. *Comm. ACM 14* (1971), 780-790.

Professional Activities

Calendar of Events

ACM's calendar policy is to list open computer science meetings that are held on a not-for-profit basis. Not included in the calendar are educational seminars, institutes, and courses. Submittals should be substantiated with name of the sponsoring organization, fee schedule, and chairman's name and full address.

One telephone number contact for those interested in attending a meeting will be given when a number is specified for this purpose.

All requests for ACM sponsorship or cooperation should be addressed to Chairman, Conferences and Symposia Committee, Seymour J. Wolfson, 643 MacKenzie Hall, Wayne State University, Detroit, MI 48202, with a copy to Louis Fiora, Conference Coordinator, ACM Headquarters, 1133 Avenue of the Americas, New York, NY 10036; 212 265-6300. For European events, a copy of the request should also be sent to the European Representative. Technical Meeting Request Forms for this purpose can be obtained from ACM Headquarters or from the European Regional Representative. Lead time should include 2 months (3 months if for Europe) for processing of the request, plus the necessary months (minimum 2) for any publicity to appear in *Communications.*

Events for which ACM or a subunit of ACM is a sponsor or collaborator are indicated by ■. Dates precede titles.

In this issue the calendar is given in its entirety. New Listings are shown first; they will appear next month as Previous Listings.

NEW LISTINGS

15 April 1978
■ **East Central SIGCSE Regional Conference,** Granville, Ohio. Sponsor: ACM SIGCSE. Conf. chm: James S. Cameron, Dept. of Mathematical Sciences, Denison University, Granville, OH 43023; 614 587-0810.

20-21 April 1978
1978 Computer Users Conference, East Texas State University, Commerce, Tex. Sponsor: East Texas State University. Contact: Donna Hutcheson, Dept. of Computer Science, East Texas State University, Commerce, TX 75428; 214 468-2954.

30 April-3 May 1978
Computers in Activation Analysis and Gamma-Ray Spectroscopy, Mayaguez, Puerto Rico. Sponsor: American Nuclear Society. Gen. chm: B. Stephen Carpenter, NBS, Activation Analysis, B118-Bldg. 235, Washington, DC 20234.

17-19 May 1978
Workshop on Petri-Nets, Erlangen, Germany. Sponsor: Gesellschaft für Informatik in cooperation with Institut für Mathematische Maschinen und Datenverarbeitung. Contact: Workshop uber Petrinetze, c/o Institut für Mathematische Maschinen und Datenverarbeitung, Universität Erlangen-Nürenberg, Martensstrasse 3, 8520 Erlangen, Germany.

19-21 July 1978
■ **Conference of Canadian Society for Computational Studies of Intelligence,** Toronto, Canada. Sponsor: Canadian Society for Computational Studies of Intelligence in cooperation with ACM SIGART, Conf. chm: C. Raymond Perrault, Dept. of Computer Science, University of Toronto, Toronto, Ont., Canada M5S 1A1.

23-28 July 1978
International Users Conference on Computer Mapping Software and Data Bases, Harvard University, Cambridge, Mass. Sponsor: Harvard University Laboratory for Computer Graphics and Spatial Analysis. Conf. chm: Allan Schmidt, 520 Gund Hall, Harvard University, Cambridge, MA 02138; 617 495-2526.

14-15 August 1978
■ **ACM SIGCSE 9th Technical Symposium on Computer Science Education,** Pittsburgh, Pa. Sponsor: ACM SIGCSE in cooperation with IEEE-CS. Conf. chm: Alfs T. Berztiss, Dept. of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260; 412 624-6458.

29 October-1 November 1978
New Orleans '78 International Data Processing Conference and Business Exposition, New Orleans Hilton Hotel, New Orleans, La. Sponsor: DPMA. Contact: Conference Coordinator, DPMA International Headquarters, 505 Busse Highway, Park Ridge, IL 60068; 312 825-8124.

5-8 November 1978
Second Annual Symposium on Computer Applications in Medical Care, Washington, D.C. Sponsor: George Washington University. Contact: F. Helmuth Orthner, Dept. of Clinical Engineering, School of Medicine and Health Sciences, 2300 K St., NW, The George Washington University, Washington, DC 20037.

15-17 November 1978
■ **Software Quality Assurance Workshop: Functional and Performance Issues,** San Diego, Calif. Sponsors: ACM SIGMETRICS, SIGSOFT, and Los Angeles Chapter. Gen. chm: A.C. (Toni) Shetler, Xerox Corp., A3-49, 701 South Aviation Blvd., El Segundo, CA 90245; 213 679-4511 x1968.

4-6 December 1978
Winter Simulation Conference, Miami Beach, Fla. Sponsors: NBS, AIIE, IEEE Systems, Man, and Cybernetics Society, ORSA, TIMS, SCS. Prog. chm: Norman R. Nielsen, Information Science Laboratory, (J-1041), SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025; 415 326-6200 x 2859.

20-22 February 1979
■ **ACM Computer Science Conference,** Dayton, Ohio. Sponsor: ACM. Conf. chm: Lawrence A. Jehn, Computer Science Dept., University of Dayton, Dayton, OH 45467; 513 229-3831.

14-16 March 1979
■ **Twelfth Annual Simulation Symposium,** Tampa, Fla. Sponsors: ACM SIGSIM, IEEE-CS,
(Calendar continued on p. 249)