1975

One Address Computers are Faster and Use Less Memory Space to Execute Arithmetic Assignment Statements

Victor Schneider

Bradford Wade

Report Number:
75-149

ONE ADDRESS COMPUTERS ARE FASTER AND USE LESS MEMORY SPACE TO EXECUTE ARITHMETIC ASSIGNMENT STATEMENTS

Victor Schneider
Bradford Wade

Computer Sciences Department
Purdue University
West Lafayette, Indiana 47907

CSD TR 149

One Address Computers Are Faster and Use Less Memory Space to Execute Arithmetic Assignment Statements

Victor Schneider

Bradford Wade

Computer Sciences Department
Purdue University
West Lafayette, Indiana
47906

Work on this paper was supported by NSF Grant GJ-31572 to Purdue University and by IBM Corporation graduate fellowships.

One Address Computers Are Faster and Use Less Memory Space to Execute Arithmetic Assignment Statements

Index Terms: computer instruction sets, machine architecture, code size minimization, minimization of execution time

C. R. Categories: 4.12, 4.22, 4.6, 6.21

Abstract

A notation is developed which permits space and time efficiency comparisons of four basic computer architectures in use today for executing Fortran-style assignment statements. From the comparisons, we discover that a suitably designed 1-address architecture (one accumulator machine) outperforms the other architectures in speed of execution and in encoded size of compiled Fortran statements. The comparisons are valid for CPU's ranging from very inexpensive designs with few registers to the most expensive designs having many registers and employing "pipelining" techniques or lookahead fetches of operands or instructions into fast cache memories.

Introduction

This paper provides some answers to questions of what sort of a computer instruction set will give best execution-time performance for simple Fortran assignment statements (or Algol or PL/1, for that matter) such as those considered in Knuth's statistical study of Fortran program characteristics (3). We compare four different computer architectures with respect to the number of bits needed to represent the object code of selected Fortran assignment statements and the execution time in microseconds for the object codes. The four architectures are

(a) The stack machines, notably the ICL KDF9 and the Burroughs 5000 and 6000 series computers,

(b) The 1-address machines, as implemented in numerous 16-bit minicomputers, such as the IBM 1130 and 1800, the Varian 70 series, and many others,

(c) The 2-address machines, most well known of which is the IBM 360/370 series, but also represented in the Data General minicomputers, and in computers made by companies like XDS and Interdata.Corporation,

(d) The 3-address machines, currently exemplified by the CDC 6000 and Cyber 70 series of computers. These machines are not strictly 3-address, in the sense that only the register-to-register operations actually involve three independent addresses (namely, two source registers and a destination register). But, since these machines are so widely used for scientific computing by universities and government research laboratories, we include them for sake of comparison.

Our method of comparison involves comparing the effects of implementing the four architectures on a selected CPU. Thus, we are not comparing Burroughs computers with CDC equipment, but rather Burroughs machines wired into CDC equipment or CDC machines wired into Burroughs equipment. Another way of looking at this is to talk about implementations of the four architectures on a CPU using emulation techniques, possibly assisted by hard wired

decoding circuits and specially designed fast cache memories. In this way, we compare apples with apples, rather than apples with coconuts or bananas.

Another point we should make is that the 1-address architecture is assumed to include "reverse subtract" and "reverse divide" operations, so that the noncommutative nature of these operations can be disregarded in our comparisons. (4) Finally, the "paper and pencil" results obtained in this study have been confirmed empirically through emulations on a microprogrammed minicomputer and through simulations and timings run on IBM 370 equipment. (6)

For standard definitions of the four architectures, the reader is referred to a textbook, such as Gear's. (1)

Notation

Let x be the number of bits used to represent the "opcode", or machine operation, portion of an instruction in any of the architectures to be compared. Then, y is the number of bits used to address source and destination registers in both the 2-address and 3-address architectures, w is the number of bits used for short addresses of operands stored in some region of main memory, and v is the number of bits used for a "long address" of operands stored anywhere at all in main memory.

If anyone reproaches us for not considering variable-length encoding of opcodes, we will respond by letting x be the average number of bits used to represent an opcode, or, for a persistent critic, we would let x be the minimum number of bits needed to represent an opcode in the encoding. The point is to select some value for x that quiets the opposition, and then show that the results still hold.

In most contemporary architectures, x varies from 5 to 8 bits, y is 3 or 4 bits, and we will endow our 2-address architecture with the same number of software registers as the 3-address version. In machines having a short addressing scheme, w is typically 8 bits, or 7 bits signed, and is often treated (unwisely, we believe) as an offset from the program counter, thus mixing alterable operands with executable instructions in the object code. The approach that we advocate is to use an implicit base register containing the (stack pointer to the) beginning address of the currently used table of program variables. Typical values for v range from 15 bits to 22 or even 24 bits. For our purposes,

$$v > x \geq w \geq y$$

a fact that we will use later in comparisons of space needed to store algorithms for different architectures.

An interesting question that arises next is "what can we say about the contribution of each portion of a software instruction to its total execution time?" In expensively designed CPU's having an abundant supply of internal registers or fast scratchpad memory, it is apparent that the operations on register data must

be included in the time estimate for executing an operation on the expensive e-CPU; namely,

$$t_e(x) = t_e(x) + 2g_e(y) = t_e(x) + 3g_e(y).$$

At the other extreme, in a very inexpensive c-CPU there may be so few registers available that the software registers are implemented as special locations in main memory. For this cheap c-CPU,

$$t_c(x) + 3g_c(y) > t_c(x) + 2g_c(y) > t_c(x).$$

A fetch to or store from a long address should take the same time in an e-CPU as the corresponding operations on a short address; i. e.,

$$f_e(v) = f_e(w).$$

For the c-CPU, however, there are usually problems caused by the inexpensive main memory used, in which the memory word width in bits is less than the word width implemented in the software instruction set. In this situation,

$$f_c(v) > f_c(w).$$

Because the c-CPU, e-CPU, and any intermediate price CPU's all emulate the same software instruction set in each architecture, all sequences of instructions fit the same number of bits in main memory for all CPU's emulating the same architecture.

Timing and Space Comparisons

Knuth's study (3) found that the simple Fortran assignment statement of the form

A = B

constituted 68% of the statically measured corpus of programs that was analyzed. Table 1 below presents the compiled code and corresponding space requirements of this statement for all four architectures.

Table 1: A = B

Stack

Addr(A)	x+v
Value(B)	x+v
Store	x
Pop	x

Space: 4x+2v
Time: 4t(x)+2f(v)

1-Address

Load B	x+v
Store A	x+v

Space: 2x+2v
Time: 2t(x)+2f(v)

2-Address

Load B in R1	x+v+y
Store R1 in A	x+v+y

Space: 2x+2v+2y
Time: 2t(x)+2f(v)+2g(y)

3-Address

Load B in R1	x+v+y
Store R1 in A	x+v+y

Space: 2x+2v+2y
Time: 2t(x)+2f(v)+2g(y)

In the Knuth study, approximately 70% of the remaining assignment statement statements (those in which arithmetic was performed) had the general form

A = B operator C.

The comparison of the four architectures for the compiled code of such a statement, with "+" taken as the operator, is given in Table 2.

Table 2: Statement A = B + C

Stack

Addr(A)	x+v
Value(B)	x+v
Value(C)	x+v
Add	x
Store	x
Pop	x

Space: 6x+3v
Time: 6t(x)+3f(v)

1-Address

Load B	x+v
Add C	x+v
Store A	x+v

Space: 3x+3v
Time: 3t(x)+3f(v)

2-Address

Load B in R1	x+v+y
Add C to R1	x+v+y
Store R1 in A	x+v+y

Space: 3x+3v+3y
Time: 3t(x)+3f(v)+3g(y)

Table 2 (continued)

3-Address

Load B in R1	x+v+y
Load C in R2	x+v+y
R3 = R1 + R2	x+ 3y
Store R3 in A	x+v+y

Space: 4x+3v+6y
Time: 4t(x)+3f(v)+3g(y)

In Tables 1 and 2, it is apparent that the 1-address machine is superior to the other architectures in terms of space and time.requirements for the two statements considered. These two statements represent 90% of all statically measured assignment statements in the Knuth study. It could be argued that the remaining 10% of assignment statements are probably executed with much greater frequency than their static fraction of all code would indicate. As an example of one such more complex statement that requires no use of temporaries to store intermediate values during the computation, consider the statement of Table 3:

Table 3: Statement A = B1 + B2 + B3 + B4

Stack

Addr(A)	x+v
Value(B1)	x+v
Value(B2)	x+v
Add	x
Value(B3)	x+v
Add	x
Value(B4)	x+v
Add	x
Store	x
Pop	x

Space: 10x+5v
Time: 10t(x)+5f(v)

1-Address

Load B1	x+v
Add B2	x+v
Add B3	x+v
Add B4	x+v
Store A	x+v

Space: 5x+5v
Time: 5t(x)+5f(v)

2-Address

Load B1 in R1	x+v+y
Add B2 to R1	x+v+y
Add B3 to R1	x+v+y
Add B4 to R1	x+v+y
Store R1 in A	x+v+y

Space: 5x+5v+5y
Time: 5t(x)+5f(v)+5g(y)

3-Address

Load B1 in R1	x+v+y
Load B2 in R2	x+v+y
R1 = R2 + R3	x +3y
Load B4 in R2	x+v+y
R3 = R1 + R2	x +3y
Store R3 in A	x+v+y

Space: 8x+ 5v +14y
Time: 8t(x)+5f(v)+14g(y)

From inspection of Tables 2 and 3, it is clear that increasing the length of the assignment statement of Table 3 still leaves the 1-address machine in the position of consuming less memory space and running as fast or faster than the other architectures. The assignment statements in Tables 2 and 3 generate no intermediate values that must be stored temporarily during a computation (5). What happens when intermediate results are generated by a computation? Consider the assignment statement of Table 4, that generates one intermediate result. (Since the 3-address machine obviously will not compete at this and further levels of complexity, we omit it from subsequent tables and discussion.)

Table 4: Statement A = B * C + D * E

Stack		1-Address		2-Address	
Addr(A)	x+v	Load B	x+v	Load B in R1	x+v+y
Value(B)	x+v	Mpy C	x+v	Mpy C by R1	x+v+y
Value(C)	x+v	Store T1	x+w	Load D in R2	x+v+y
Multiply	x	Load D	x+v	Mpy E by R2	x+v+y
Value(D)	x+v	Mpy E	x+v	Add R2 to R1	x+ 2y
Value(E)	x+v	Add T1	x+w	Store R1 in A	x+v+y
Multiply	x	Store A	x+v		
Store	x			Space: 6x+5v+6y	
Pop	x	Space: 7x+5v+2w		Time: 6t(x)+5f(v)+6g(y)	
Space: 9x+5v		Time: 7t(x)+5f(v)+2g(w)			
Time: 9t(x)+5f(v)					

At this point, the 1-address architecture begins to lose ground. In particular, we have to provide a short-address format for storage of intermediate values in order to retain the 1-address space advantage. We also have to note that a modified stack architecture that provides a Store(A) instruction to store the top of the stack in A and pop the stack definitely takes less space and executes the statement in Table 4 in time comparable to the 1-address machine. Another point to note is that the 2-address machine will execute this statement more rapidly than the others on an expensive CPU in which $g(y)=0$ and only requires $6y-x$ more bits to encode than the modified stack machine.

It becomes interesting then to see what happens when more than one intermediate result is generated in the course of executing

an assignment statement. The example that we use is of an assignment statement written so as to be impervious to the complexity-reducing manipulations of an optimizing compiler (5). Because of its special form, it tends to penalize the 1-address architecture (there are no sequences such as A*B*C*D that would favor the machine) in favor of other architectures.

Table 5: Statement A = ((B*C + D*E) * (F*G + H*J))

Stack		1-Address		2-Address	
Addr(A)	x+v	Load B	x+v	Load B in R1	x+v+y
Value(B)	x+v	Mpy C	x+v	Mpy C by R1	x+v+y
Value(C)	x+v	Store T1	x+w	Load D in R2	x+v+y
Multiply	x	Load D	x+v	Mpy E by R2	x+v+y
Value(D)	x+v	Mpy E	x+v	Add R2 to R1	x +2y
Value(E)	x+v	Add T1	x+w	Load F in R2	x+v+y
Multiply	x	Store T1	x+w	Mpy G by R2	x+v+y
Add	x	Load F	x+v	Load H in R3	x+v+y
Value(F)	x+v	Mpy G	x+v	Mpy J by R3	x+v+y
Value(G)	x+v	Store T2	x+w	Add R3 to R2	x +2y
Multiply	x	Load H	x+v	Mpy R2 by R1	x +2y
Value(H)	x+v	Mpy J	x+v	Store R1 in A	x+v+y
Value(J)	x+v	Add T2	x+w		
Multiply	x	Mpy T1	x+w	Space: 12x+9v+15y	
Add	x	Store A	x+v	Time: 12t(x)+9f(v)+15g(y)	
Store	x				
Pop	x	Space: 15x+9v+6w			
		Time: 15t(x)+9f(v)+6g(w)			

Space: 18x+9v
Time: 18t(x)+9f(v)

On the assumption that w=y, the 2-address machine requires 9y-3x more bits to encode the Table 5 algorithm, and takes 9g(y)-3t(x) microseconds more time to execute than does the 1-address version. For a very inexpensive CPU, in which g(y) > t(x)/3, the 1-address machine is slightly faster; but, for all other situations, the 2-address architecture is uniformly superior in speed of execution. The space estimate only slightly favors the stack architecture for this algorithm.

As a final example, we can consider the statement in Table 6 that calls for only one intermediate value, but "favors" the 1-address machine by providing more than the minimum of operations to force an intermediate value in the object code:

Table 6: Statement A = B*C*D + E*F

Stack		1-Address		2-Address	
Addr(A)	x+v	Load B	x+v	Load B in R1	x+v+y
Value(B)	x+v	Mpy C	x+v	Mpy C by R1	x+v+y
Value(C)	x+v	Mpy D	x+v	Mpy D by R1	x+v+y
Multiply	x	Store T1	x+w	Load E in R2	x+v+y
Value(D)	x+v	Load E	x+v	Mpy F by R2	x+v+y
Multiply	x	Mpy F	x+v	Add R2 to R1	x +2y
Value(E)	x+v	Add T1	x+w	Store R2 in A	x+v+y
Value(F)	x+v	Store A	x+v		
Multiply	x			Space: 7x+6v+8y	
Add	x	Space: 8x+6v+2w		Time: 7t(x)+6f(v)+8g(y)	
Store	x	Time: 8t(x)+6f(v)+2g(w)			
Pop	x				
Space: 12x+6v					
Time: 12t(x)+6f(v)					

Here again, if we allow w=y (a limiting case), then the 1-address machine requires 6y-x less bits of space than the 2-address machine, an improvement in space usage over the Table 4 example, and the execution time comparisons improve slightly for the case of the inexpensive CPU.

Conclusions

This paper has demonstrated that, for all but the most expensive CPU's, a CDC-style 3-address architecture is least efficient for executing simple assignment statements, both in terms of bits needed to encode the algorithms and times of execution. A 2-address machine compares most advantageously in a CPU having an operand cache-i. e., high-speed hardware for prefetching operands. With such hardware, the 2-address machine is uniformly as fast or faster than the competition. For a simpler CPU, or one with an instruction cache, however, the 1-address architecture will outperform its rivals in speed of execution, and will on the average require fewer bits to encode its assignment statements. The stack machine, even in a modified version that performs 1-address stores from the stack into main memory, offers no advantages, either in execution time or in bits required to encode algorithms.

References

1. Gear, William C., Computer Organization and Programming. McGraw-Hill Book Co., New York, 1969.

2. Hager, K., "Die Bewertung einiger Rechnerkerntypen fuer das Verarbeiten von arithmetischen Ausdruecken," Elektronische Rechenanlagen 13, 6 (Dec., 1971), pp. 241-249.

3. Knuth, D. E., "An Empirical Study of FORTRAN Programs," Stanford University, Computer Science Department Report No. CS-186.

4. Lindsey, C. H., "Making the Hardware Suit the Language," in ALGOL 68 Implementation (J. E. L. Peck, ed.), Amsterdam: North-Holland Publishing Co., 1971.

5. Schneider, Victor B., "On the Number of Registers Needed to Evaluate Arithmetic Expressions," BIT 11 (1971), 84-93.

6. Wade, Bradford W., A General-Purpose High-Level Language Machine for Minicomputers, Doctoral Dissertation, Purdue University, August, 1975.