AN ARCHITECTURE WITH MANY OPERAND REGISTERS TO EFFICIENTLY EXECUTE BLOCK-STRUCTURED LANGUAGES*

Published in: Dannenberg, “An Architecture With Many Operand Registers to Efficiently Execute Block Structured Languages,” in *Proceedings of the 6th Annual Symposium on Computer Architecture,* Philadelphia, PA, April 1979. New York: ACM, 1979. pp. 50-57.

Roger B. Dannenberg

Department of Systems Engineering,
Computer Engineering, and Information Sciences
Case Western Reserve University
Cleveland, Ohio 44106

Abstract

Register allocation schemes are presented that effectively use many registers in the execution of block-structured languages. Simulation statistics for a machine with many registers and a conventional architecture are compared. The results indicate that the average operand access time and the required memory bandwidth of conventional machines can be significantly reduced. The implications of the register allocation schemes for machine architecture are discussed.

1. Introduction

Recent advances in semiconductor technology have made large register sets feasible. By "register set," we mean a memory with an access time on the order of one tenth that of primary memory; by "large," we mean hundreds or thousands of words. Methods of using many registers to speed up the execution of block-structured languages with potentially recursive procedures are presented. These methods require special hardware in addition to many registers. Finally, results from a simulation study are presented and discussed.

1.1 Background

Many processors have a small set of registers, numbering 4 to 16. These registers can be used to contain variables which are repeatedly used, thus saving memory accesses and decreasing the average operand access time. Unfortunately, much of this savings is lost in the overhead of loading and storing registers. Studies have found that about 30% to 40% of all instructions executed are simple moves between registers and memory. Furthermore, measurements have shown that compilers are unable to effectively use more than a few registers.[1,2,3] This is partially due to the fact that compilers almost invariably save and restore registers on subroutine calls.[4] A large number of registers are provided in the CRAY-1 computer. A method for efficient use of these registers is described in [5] and is similar to the techniques to be described. This paper is concerned with the problem of effectively using registers to reduce the average operand access time. The reduction of primary memory bandwidth is also discussed. Our approach only deals with the speedup of data references; instruction fetches are not considered.

Let us first consider the traditional method of run-time storage administration for a block-structured language. Because procedures may be entered recursively, memory for variables within each procedure is allocated from a common stack. When entered, a procedure first allocates an activation record on the stack (Figure 1). The activation record provides storage for all the fixed-length variables declared within the procedure. Data structures whose lengths are not known at compile time are separately allocated on the stack and are accessed through fixed-length descriptors in the activation record.

In order to access activation records themselves, a set of pointers called a display is used. The display holds addresses of currently accessible activation records. Figure 2 illustrates an instance of the execution of a program. Procedure Q has been called recursively, so two activation records for Q are on the stack. The semantics of block-structured

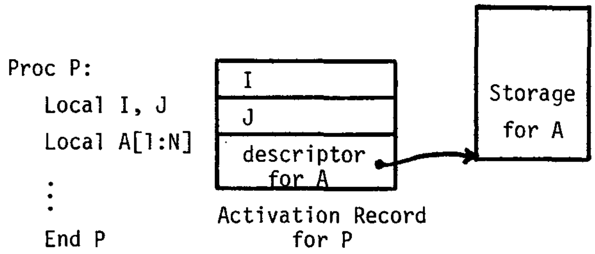


Figure 1. A procedure and its activation record.

*Work partially supported by NSF grant SPI-7821198.

```
Proc P:
   Proc Q:
      Proc R:
         body of R
         End R
      body of Q
      End Q
   Proc S:
      body of S
      End S
   body of P
   End P
```

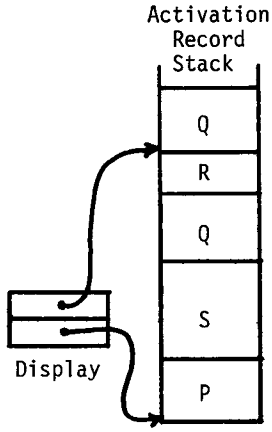


Figure 2. Activation records and display during an execution of procedure Q

languages specify that procedure Q can only access the most recent activations of variables declared in P or Q. Therefore, the display only contains pointers to the most recently allocated activation records for these procedures. Various schemes are used to maintain the display and manage the stack. Here, we are only concerned with the manner in which variables are stored and accessed.

To access a variable in this conventional scheme, two address fields are required. The first selects a display pointer. To this pointer is added the second address field which is the offset of the variable in the activation record. Finally, the variable is fetched from the computed address. Assuming that the display is maintained in registers, accessing a simple variable requires the fetching of two offsets, a register access, an addition, and a data-memory reference. This process is illustrated in Figure 3. Several variations of this scheme are used. They differ primarily in the way the display is maintained and accessed.

One variation always maintains the display in high-speed registers. The registers are updated on procedure entry and exit to maintain the correct display. Now, suppose instead of maintaining pointers to activation records, we put into registers parts of the activation records themselves. This would eliminate an addition and a memory reference for some variable accesses, leaving only a register access. Only one offset would be required to address the variable. Of course, the overhead of maintaining this new "display" is higher than before, and more registers are required.

2. Using Many Registers

2.1 Register Allocation

To evaluate the efficiency of this concept, several methods of maintaining the registers will first be examined. In general, the size of an activation record could be much too large to be kept entirely in registers. This is the case when large structures (e.g. arrays or records) are part of the activation record. The approach taken here is a simple one: keep all simple variables and structure descriptors in registers. Structures themselves are kept in primary memory on a stack. From here on we will use "variables" to refer to both simple variables and structure descriptors, but not structures. The term "activation record" will refer to the set of variables in a procedure. We will also assume that separate activation records are not created for blocks within procedures, but rather the procedure's activation record contains space for all variables declared inside the procedure. This scheme is commonly used by conventional compilers because the overhead of creating a new activation record is unnecessary for block entry.[6]

It is obvious that our method will require many registers. Recent advances in semiconductor technology have made this feasible. Several manufacturers are now producing 4k bit integrated circuit memory chips with access times of less than 100ns.[7] One thousand 16-bit registers could be implemented with four chips. Information in [8] indicates that one thousand registers is more than adequate for a large program.

While the basic concept is to keep all accessible variables in registers, a

1) fetch address pair (a,b)
2) fetch Display[a]
3) add offset b
4) fetch memory word at the address (Display[a] + b)

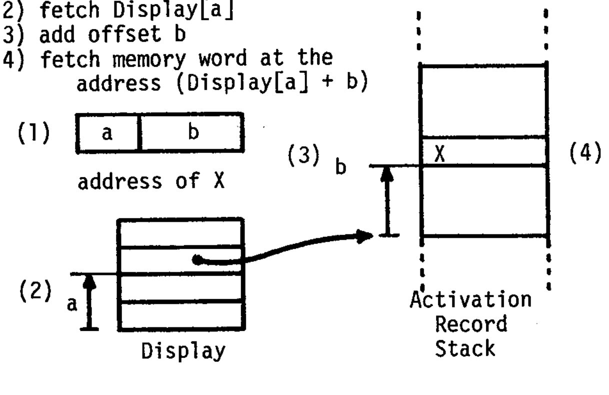
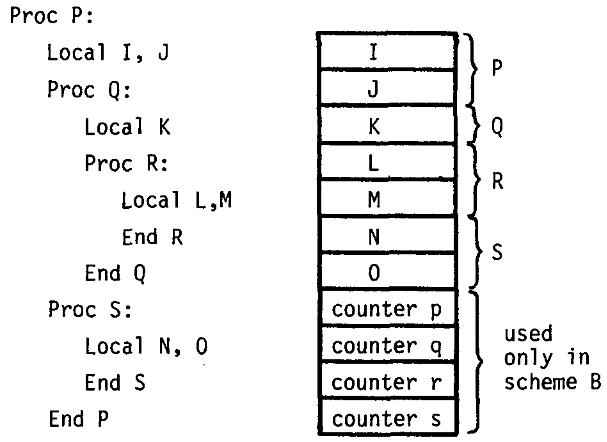


Figure 3. Conventional address calculation to access a variable in a block-structured language.

number of variations are possible. Several schemes are presented which differ in the way registers are statically and dynamically allocated.

2.1.1 Scheme A. Scheme A (Figure 4) statically allocates registers for activation records such that no registers are shared by activation records. At run time, only the most recent activation record for each procedure is kept in registers. When a procedure is entered, the contents of the registers are copied into a stack in primary memory. When the procedure is exited, the contents are restored from the stack. Registers are copied so that register contents are not destroyed when a procedure is recursively called. Because structures are allocated in primary memory, only short descriptors need be copied to provide for recursion.

For example, in Figure 4, suppose R is called. R saves L and M in case these hold values from another activation of R. When R returns, L and M are restored. Notice that during the execution of R, all accessible variables (I,J,K,L,and M) are in registers.



```
Proc P:
   Local I, J
   Proc Q:
      Local K
      Proc R:
         Local L,M
         End R
      End Q
   Proc S:
      Local N, O
      End S
   End P
```

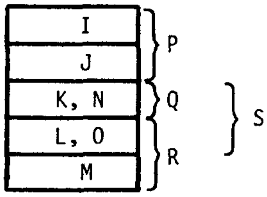
activation records do not overlap

Figure 4. Allocation of registers for schemes A and B.

2.1.2 Scheme B. Scheme A can be made more efficient by only saving registers when a procedure is recursively entered. An extra register is associated with each procedure as a recursion counter. The counter is initialized to zero and incremented upon procedure entry. If the result is greater than one, the registers (excluding the counter) are saved in primary memory as before. Otherwise, the registers do not contain any useful information and need not be saved. On procedure exit, the counter is decremented. If it is non-zero, registers are restored by copying values from the stack; otherwise, no action is taken. This will be refered to as scheme B.

2.1.3 Scheme C. A disadvantage of schemes A and B is that the number of registers needed is approximately the total number of variables in all procedures. Scheme C introduces a strategy which reduces this number, but requires as much copying as scheme A. In scheme C, variables within a procedure are assigned to successive registers. Given that register i is the last register used by procedure P, then register i+1 is the first register used by each procedure declared in P. Thus, many variables may be associated with a single register (Figure 5). Since no two variables accessible to a given procedure are assigned to the same register, it is still possible to keep all accessible variables in registers. The copying rule of scheme A is used: save all registers upon entry and restore them upon exit. Recursion counters are not used because registers are shared among different activation records.

Consider the previous example where R is called. Again, R saves registers for L and M. In this case, the registers saved may contain parts of other activation records (e.g. variable O in procedure S), but only variables which are inaccessible to R are saved. The register contents are restored when R returns to its caller.



Note that activation records overlap.

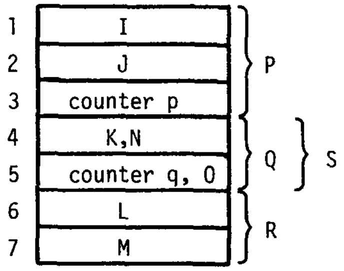
Figure 5. Allocation of registers for scheme C. (same program as in figure 4)

2.1.4 Scheme D. Although similar to scheme C, scheme D uses a different copying rule. Consider the behavior of scheme C when a procedure repeatedly calls a nonrecursive function (or procedure) declared within the procedure. On each function call, the function must save all the registers corresponding to its activation record. The same values are saved and restored each time the function is called. If instead those registers are saved when the procedure is entered, there is no need to repeatedly save them when the function is called. Scheme D accomplishes this with its copying rule which guarantees that whenever control passes from one nesting level to the next, no registers are saved. Saves are performed only when procedures at the same or outer levels are called. A variable that serves as a counter is added to each procedure containing other procedure declarations (Figure 6). The counter is shared by the procedures declared in the same enclosing procedure. A counter is also added to the level Ø (global) activation record and initialized to zero. When a procedure at level i is entered, the counter at level i-1 is incremented. If the result is not one, all registers used by the procedure and by any enclosed procedures are saved on the stack in primary memory. If the counter equals 1, no copy is performed. Finally, if there is a counter at level i, it is initialized to zero. To exit a procedure at level i, the counter at level i-1 is decremented and registers are restored if appropriate.

For example, suppose Q calls procedure R. Q has already saved registers 4 through 7 (Figure 6) and initialized counter q to zero. When R is called, counter q is incremented; the result indicates R does not need to save registers for its variables L and M. If R then calls itself, counter q is incremented again. The result is not one (1), indicating that registers 6 and 7 are in use and must be saved. Thus, recursion is safely handled, but R does not save registers when called by Q.

2.2 Parameter Passing

Parameters may be passed in a variety of ways; the best mechanism will depend upon the intended semantics (e.g. call-by-value) and features of the instruction set.



When registers are saved,
P saves registers 1 through 7
Q saves registers 4 through 7
R saves registers 6 through 7
S saves registers 4 through 5

Figure 6. Allocation of registers for scheme D. (same program as in figure 4)

2.2.1 Call-By-Value. Consider an architecture which uses an evaluation stack for its arithmetic computations, and assume non-structured parameters are passed by value. Structures are passed by reference. The calling procedure simply evaluates parameter expressions and leaves them on the stack. In the case of structured parameters, the corresponding descriptor is pushed on the stack. If necessary, the called procedure first saves the appropriate registers. Then parameters are popped from the stack into registers in the procedure´s activation record. This is the mechanism used in our simulation study. The evaluation stack is distinct from the activation record stack and is implemented with a small number of fast registers.

Call-by-value/result can be implemented by returning the results on the evaluation stack in a similar manner.

2.2.2 Call-By-Reference. Two problems are encountered with call-by-reference parameters. First, a reference parameter must exist in a single addressable location for the duration of the procedure call. Secondly, if the actual parameter is a global variable, we must be able to refer to the same location as a global or as a parameter. (This is known as aliasing.) To do this, we propose associating a bit with each register indicating "reference" or "value". When the bit indicates "value", the content of the register is the value of a simple variable or structure descriptor. "Reference" indicates that the value is at the primary memory location addressed by the contents of the register. When an instruction accesses a register and the bit indicates "reference", a memory fetch is automatically initiated to retrieve the value. Special instructions are provided which ignore the bit, so that references can be manipulated.

An instruction, REF, takes a register with a "value" and pushes it onto the structure memory stack, replacing the register contents with the address of the value and setting the bit to "reference" (Figure 7). Another instruction, UNREF, performs the inverse operation. Bits are initialized to "value" when the activation record is allocated at run-time.

To pass the variable in register i as a reference parameter, we execute "REF i" which puts the value into a location which is fixed for the duration of the procedure call. Register i now has an address which is passed to the procedure. When the procedure returns, "UNREF i" can be used to restore the variable value to register i and set bit i back to "value". Suppose register i holds a global variable used by the called procedure. Any instruction which accesses register i will find bit i set to "reference", and the operation will be directed to the memory location where the value is stored. Thus, aliasing is correctly handled.

If aliasing is not allowed, the simpler value/result scheme is equivalent to call-by-reference.

2.3 Implementation

2.3.1 Block Transfers. All of these schemes can require copying blocks of registers into a stack and back again; single instructions should be provided to effect these transfers. The transfers could be combined with subroutine call instructions. Registers can be saved on the same stack that structures are allocated on. Alternatively, registers can be saved on a separate stack. If this approach is used, the stack is accessed in a strict first-in, first-out manner. No references are made to data below the top of the stack. This allows the memory stack to be buffered by a simple hardware scheme. Because the average number of parameters and local variables is small [9], a small buffer should greatly reduce the number of memory references due to saving and restoring registers, although measurements presented below indicate that this may not make up a significant part of the total memory references.

2.3.2 Instruction Set. Aside from the special save and restore instructions, conventional instruction sets are suitable for the proposed architecture. The instruction set will rarely address memory directly, so most instructions will not need large address fields. Register-to-register operations may be provided, or operations may be based on a single accumulator or an evaluation stack. Some combination of the above may also be used.

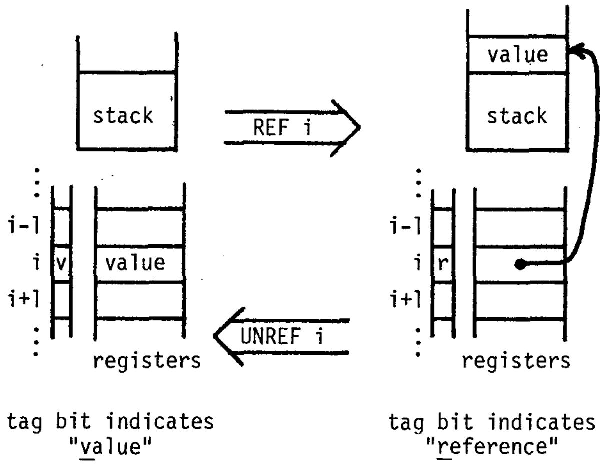


Figure 7. The operation of REF and UNREF instructions.

It is a property of the described register allocation schemes that the run-time address of a variable in a register is known at compile time. Therefore, no run-time address translation need be used when accessing variables. Operand access time is decreased because no address calculations are needed. However, there are reasons for using address translation. First, data in [9] indicates that few address bits are needed on the average if variables are addressed relative to pointers to local and global activation records. A set of 1024 registers would require 10 bits to directly address all registers. A significant savings might be obtained if registers were addressed relative to one or more pointers, initialized on procedure entry. Secondly, address mapping can allow the registers to be partitioned among several users in a multiprogramming environment. This should allow context-switching without the overhead of saving many registers.

2.3.3 Interrupts. Interrupts can be handled in the same way as ordinary subroutine calls. The interrupt routine must allocate registers for its activation area, but a complete context swap is not necessary. Various protection and memory mapping schemes are possible to guard against interference between processes.

3. Evaluation

3.1 Terminology

To evaluate this type of architecture, measurements of program behavior are used. We would like to measure the reduction of average operand access time as a result of register usage. Three parameters affect this:

1) Save/Restore overhead: The ratio of memory references due to saves and restores to total memory references.
2) Register usage ratio: The ratio of register accesses to total storage accesses (exclusive of saves and restores).
3) Register speed: The ratio of register access time to memory access time.

To make the definitions of the first two parameters more precise, let R be the total number of bytes referenced in registers, and let M be the total number of bytes referenced in primary memory. Let r be the number of register bytes referenced to perform saves and restores, and let m be the number of primary memory bytes referenced due to register saving and restoring. Both instruction fetches and data references are included in m. The save/restore overhead is then given by:

$$\text{Save/restore overhead} = \frac{m}{M}$$

and the register usage ratio is given by:

$$\text{Register usage ratio} = \frac{(R-r)}{(R-r)+(M-m)}$$

The register usage ratio is analagous to the hit ratio of cache memory systems. Both give the ratio of storage references satisfied by high-speed storage to the total storage references, ignoring references that are made to maintain data in the high-speed store. The register usage ratio is independent of the allocation scheme used provided that all accessible variables are kept in registers.

3.2 Methodology

A simulation has been performed to measure the first two parameters. An interpreter was used to emulate a 16-bit processor with many registers. The registers of this processor are 16 bits wide, and a 16-bit-wide evaluation stack is used for virtually all arithmetic operations. Primary memory is byte-addressable, and most instructions are 8 bits long. The most frequently executed instructions, push-register and pop-register, are encoded in single bytes, including the register address. The tag bits proposed to allow call-by-reference parameters were not included in the simulation.

The interpreter is instrumented to record the number of memory accesses for both the instructions and data. Register accesses and procedure calls are counted. Separate counts are kept for each procedure. References to the evaluation stack are not included in the count for register accesses. The intent is to measure "useful" register references, not the reference overhead of the stack mechanism. For example, "PUSH A; PUSH B; ADD; POP C" references only 3 registers but the stack is accessed 6 times. On the other hand, if a three-address instruction format were used, i.e. "ADD A,B,C", no stack references would be made. Given the number of calls on each procedure, we can compute the number of register saves and restores used by schemes A and C. If counts are obtained for recursive calls, we can calculate the save/restore overhead of scheme B. The particular procedures measured are not recursive, so the save/restore overhead associated with scheme B in our results is just the overhead due to fetching save/restore instructions. These instructions update the recursion counters, but this adds an insignificant overhead. In general, the overhead increases roughly in proportion to the number of recursive calls. By counting the number of calls to locally declared (nested) procedures, the save/restore overhead of scheme D can be calculated.

4. Results and Discussion

Two programs, an assembler and an editor were measured. Both of these programs are written in a block-structured language and compiled by hand. Very little optimization was performed in the translation. A total of 654,855 instruction were executed.

4.1 Save/Restore Overhead

Figure 8 summarizes the dynamic program measurements. Static counts of the number of registers used by each scheme for each program are also presented in Figure 8. Schemes A and B use significantly more registers than either C or D because the latter two allow activation records to overlap. Figure 9 presents the

Measurements for Programs ASM and ED

Parameter	ASM	ED	total
Number of Instructions Executed*	321766	333089	654855
Number of Instruction Bytes Fetched*	574985	594555	1079540
Data Bytes Referenced In Primary Memory*	33959	53616	87575
Number of Registers Referenced*	124701	150293	274994
Number of Procedure Calls	2444	1680	4224
Number of Calls to Nested Procedures	807	1400	2207
Average Number of Registers in Activation Records (static)	3.2	2.9	3.1 (av)
Average Number of Registers in Activation Records(dynamic)	2.6	3.3	3.0 (av)

*Not including save and restore operations.

Total Registers Used

scheme program	A	B	C	D
ASM	71	93	22	23
ED	35	47	15	16

Figure 8. Program Measurements.

computed save/restore overhead and register usage ratio parameters for schemes B, C, and D. Scheme A is omitted because it is identical to scheme C in all respects except the number of registers used is greater. These figures include 6 bytes of instructions per call to specify which registers are to be saved and restored. The largest save/restore overhead measured is 6.27%. We would expect this number to be quite high since our purpose is to use a large amount of copying to keep variables in registers. Lunde remarks in [4] that for a particular "highly subroutine-structured large program" running on a DECsystem10, 9% of the instructions executed are used to save and restore registers. Although this figure is not presented in terms of memory bandwidth, it indicates that conventional machines can suffer from equally large (if not greater) save/restore overheads.

The average overhead of scheme B, 1.94% is the lowest measured, however, this scheme uses significantly more registers than schemes C or D. These have higher average overheads of 5.56% and 4.45%, respectively. Scheme C is the simplest to implement, but has the highest measured overhead. Although scheme D had a lower measured overhead than C in the simulations, programs can easily be written where this is not the case. Simulations of a broader class of programs should be made to fully evaluate these schemes.

It is interesting to compare the register allocation schemes presented here with conventional register allocation techniques. As mentioned in the introduction, studies have shown compilers are unable to effectively use more than 4 to 10 registers in conventional architectures.[1,2,3] The method presented here allows many registers to be used. Furthermore, the registers are mainly used to hold program variables and structure descriptors. No registers are used to address activation records since these are kept in registers at fixed locations. Conventional register allocation algorithms are quite complex, making efficient compilation difficult and time-consuming. Algorithms that allocate registers for any of the schemes described here are trivial by comparison.

Save/Restore Overhead -- ratio of save/restore memory references to total memory references (in percent).

program	scheme: B	C	D
ASM	2.35%	6.27%	5.50%
ED	1.53%	4.78%	3.40%
average	1.94%	5.56%	4.45%

Register Usage Ratio -- ratio of register references to total storage references, excluding saves/restores (in percent).

program	register usage ratio
ASM	29.06%
ED	31.68%
average	30.37%

Figure 9. Results of simulations.

4.2 Reduction of Memory Accesses

Architectures may also be compared on the basis of reduction of primary memory accesses as a result of register usage. Several studies of instruction utilization have been reported [1,10], but it is difficult to use these reports to compare the register usage of two architectures. Instructions are used to process data and to access data. Because registers hold both addresses and "useful" data, instruction traces indicate that most operands are found in registers, even if few of these operands correspond to program variables. Lunde suggests computing a "D-ratio" [4] as a measure of instructions used in data-structuring. An alternative is to compute ratios of primary memory references that access data to the total of data accesses and instruction fetches. We will call this the data reference ratio. Let D be the number of data bytes referenced in primary memory, and let I be the number of bytes referenced for instruction fetching. If M is the total number of primary memory bytes referenced, then

$$M = I + D$$

and

$$\text{Data reference ratio} = \frac{D}{M}$$

It is assumed that a high ratio indicates that few operands are retrieved from registers. Results in [4] indicate that this ratio for the DECsystem10 varies from about 0.28 to 0.36. A value of about 0.36 is obtained from the Gibson mix [10], and 0.45 for th IBM 360.[11] The average obtained from our simulation is only 0.10. This figure includes memory references for saving and restoring registers in the worst case (scheme C). This ratio is small because th local variables are kept in registers, reducing memory accesses for data. This demonstrates that the use of many registers can significantly decrease references to memory for data. It follows that the average access time for operands is lower than in a conventional architecture. One might ask if the ratio is small because the instruction set is not efficiently encoded, increasing the memory accesses for instructions. Because registers are addressed with short addresses, and because indexing is only done to access arrays (not activation records), we feel this is not the case.

4.3 Usage Ratio

The register usage ratio is computed from the simulation and tabulated in Figure 9. The average indicates that registers account for about 30% of total storage references. This figure is quite high considering that registers are essentially used only for program variables. For example, execution of the program statement "I := J + K" would result in 3 registers (6 bytes) being referenced. A register usage ratio computed for a conventional architecture would not have the same meaning since registers are commonly used for address calculations and holding temporary results. Our measurements indicate that the access time of a significant fraction of storage references can be reduced through the use of many registers.

Other methods of reducing effective operand access times include operand prefetching and cache memories. The IBM 360 model 91 is an example of a machine that prefetches instructions and data.[11] This method is expensive to implement because of the data-dependent nature of control. Special hardware is required to keep track of instances of variables. However, the prefetching of only instructions might be useful in and architecture with many registers.

4.3.1 Comparison With Cache Memories. The cache memory offers fast access to instructions and data. This can result in higher performance than that of other schemes, but at a higher cost than a large register set. Thus, the use of many registers might be most cost effective on small computers. In general, a cache memory is slower than registers because of the associative search time. Furthermore, a cache cannot take advantage of program structure to reduce memory accesses, or to prefetch operands. Our technique also eliminates a level of indirection to access variables, which decreases access time. In a very fast computer, the two techniques might be combined, using many registers for variables and a cache for other storage.

5. Conclusion

The use of many registers can significantly decrease the average operand access time as compared to conventional register machines. Simple register allocation schemes can be used to effectively utilize many registers. Simulation results indicate that all accessible simple variables and structure descriptors can be maintained in registers with very little overhead. References to registers account for about 30% of all storage references, resulting in a lower primary memory bandwidth requirement.

Acknowledgements

The comments and criticisms of Professor Alexander Thomasian throughout the development of this paper are gratefully acknowledged.

References

[1] Lunde, A. "Empirical Evaluation of Some Features of Instruction Set Processor Architectures," CACM Vol. 20, No. 3 (Mar 1977), pp. 143-153.

[2] Yuval, G. "Is Your Register Really Necessary?," Software -- Practice and Experience, Vol. 7 (1977), p. 295.

[3] Yuval, G. "The Utility of the CDC 6000 Registers," Software -- Practice and Experience, Vol. 7 (1977), pp. 535-536.

[4] Lunde, A. "More data on the O/W ratios, A note an a paper by Flynn," Computer Architecture News, Vol. 4, No. 1 (Mar 1975), pp. 9-13.

[5] Baskett, F. "More on Microprocessors of the Future," Computer Architecture News, Vol. 6, No. 5 (Dec 1977), pp. 14-17.

[6] Gries, D. Compiler Construction for Digital Computers, John Wiley & Sons, Inc., New York, 1971, pp. 193-211.

[7] Smith, S. and Garen, E. R. "Technology Status Report On Recent NMOS Processes," Computer Design, Aug. 1978, pp. 160-162.

[8] Wilner, W. T. "Bourroughs B1700 Memory Utilization," AFIPS FJCC Proc. Vol. 41, Part 1 (1972), pp. 579-586.

[9] Tannenbaum, A. S. "Implications of Structured Programming for Machine Architecture," CACM Vol. 21, No. 3 (Mar 1978), pp. 237-246.

[10] Gibson, J. C. "The Gibson Mix," Rep. TR 00.2043, IBM Systems Development Div., Poughkeepsie, N.Y., 1970.

[11] Anderson, D. W., et. al., "The IBM System/360 Model 91: Machine Philosophy and Instruction Handling," IBM Journal of Research and Development, Vol. 11, No. 1 (Jan 1967), pp. 8-24.