DOCID: 3928891

Approved for release by NSA on 12-01-2011, Transparency Case# 63852

STATUTORILY EXEMPT

RISC Does Windows

An intriguing trait of several Reduced Instruction Set Computer (RISC) based architectures is Multiple Overlapping Register Windows (MORW). This technique has been referred to as a hardware solution for managing large numbers of registers. This paper is an in-depth discussion of Multiple Overlapping Register Windows, their occurrence in current RISC processors, and alternatives to them.

INTRODUCTION TO RISC AND REGISTER FILES

Recently there has been a great deal of activity within the field of Reduced Instruction Set Computer (RISC) processors. Most people are used to the concept of Complex Instruction Set Computer (CISC) processors, where a large instruction set provides the programmer with just about any instruction that may ever be needed. A basic assumption made in CISC designs is that compiler developers require these large instruction sets to easily generate efficient code. This assumption may not be a valid one and may serve only to cause unnecessary hardware complexity. It has been stated that compiler writers can only make use of about 30 percent of the instructions in these large instruction sets [6]. The RISC philosophy is to keep the hardware very simple, yet relatively optimized to the applications that it will be running, providing only those instructions that can be used efficiently. A major benefit of this is that the complex control mechanisms required to implement a CISC processor can be eliminated, allowing the processor in most cases to fit easily within one Very Large Scale Integrated Circuit (VLSI) chip. The impressive result is that the simple control circuitry and the regular structure of the rest of the RISC processor allows the RISC design to easily *scale* to smaller chip dimensions. This scalability permits a RISC processor to rapidly take advantage of technological advances.

A large part of the RISC design process is determining a relatively small set of instructions that covers the needs of the applications that will run on the processor. These instructions must also be easily decodable to enable a simplified control circuit. The goal is for each instruction to require only one fixed length instruction cycle to complete. This can be achieved by permitting only *load* and *store* instructions to access memory and requiring all other instructions to use registers as their operands. Consequently, RISC designs tend to require a large number of CPU registers (32 to 500+). These registers are normally presented in some form of register file that can be accommodated due to the space saved with a simplified control circuit.

Register files can be looked upon as flexible mini-caches. They are flexible because they are not restricted to being mapped onto memory in any specific way, but cache-like in that the elements are quickly accessible by the processor. Register files differ from cache systems in that register files are generally smaller than caches. The benefit here is that the smaller register file addresses can be encoded within a few bits in the instruction, resulting in quicker address decoding in hardware.

The RISC designers must decide how to manage this fairly large register file. One solution is to let the compilers take care of the register allocation problem. This is a reasonable decision but leads to a more complex compiler. Another solution is to reorganize the register file, allowing hardware to take care of this problem. A register file with Multiple Overlapping Register Windows (MORW) is one reorganization technique.

REGISTER FILES WITH MULTIPLE OVERLAPPING REGISTER WINDOWS

The management of register files with MORWs is a hardware solution for handling a large register file. In this technique, the register file is broken up into equal sized windows, of which only the current window is visible to the processor. These windows are not completely independent; they overlap to provide partial sharing of registers between windows. This allows the processor to switch to the next or previous window without losing all of its old registers. For example, consider figure 1. Window N has registers local

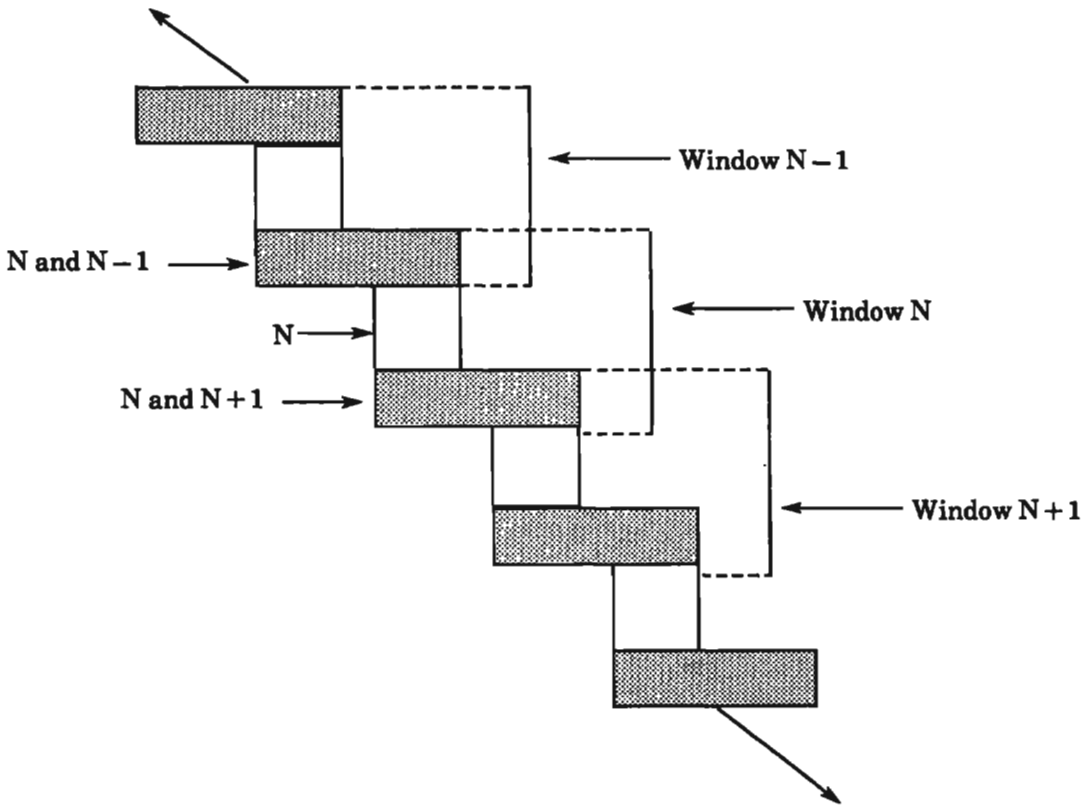


Fig. 1. Register Sharing between Multiple Overlapping Register Windows

only to itself, registers shared with N − 1, and those shared with N + 1. If the current window being used by the processor is N and a value is placed into a register shared with N + 1, then this value can be accessed in window N or in N + 1 if the processor switches its current window to N + 1. Due to its finite nature, the register file can be viewed as a large wheel with its first and last windows sharing registers. (See figure 2.)

In several RISC architectures a register file structure similar to this is used for a more specific reason. Every time a procedure or function call occurs, the parameters are placed in the registers shared with the next window and that window becomes the current window. The procedure or function then has its parameters in a fresh register window. When the procedure or function is done, results are stored in the registers shared with the previous window and the previous window becomes the current window. In this model, procedure and function CALL/RETURN can be viewed as the register window wheel rolling back and forth. But with a structure such as that in figure 2, the CALL/RETURN

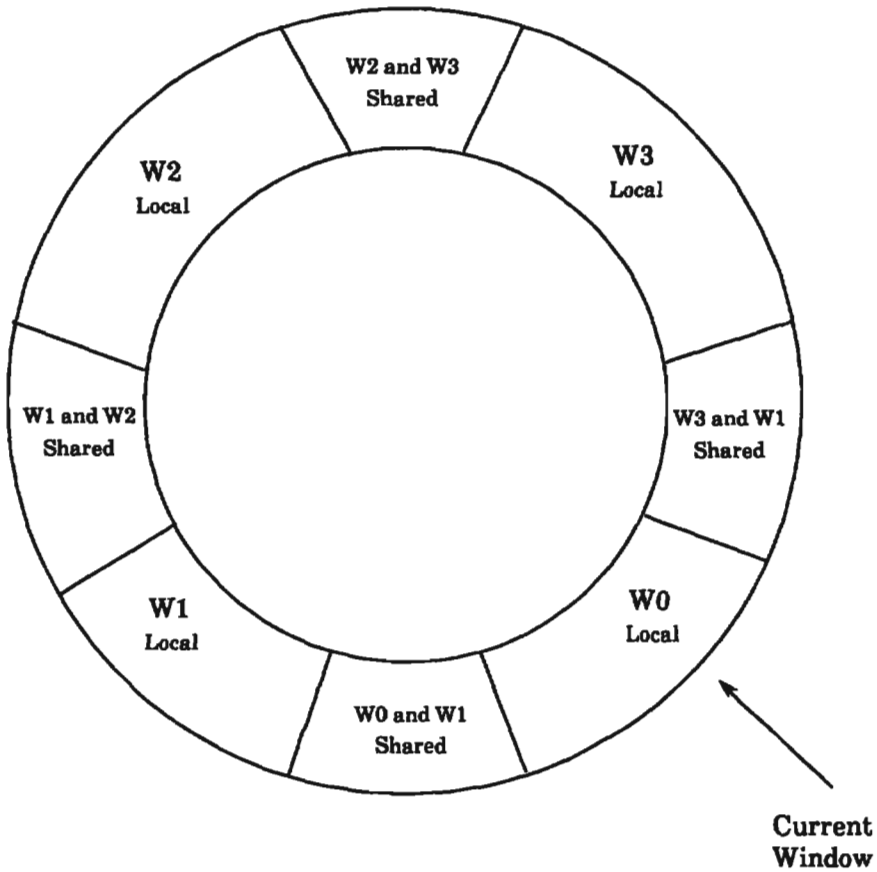


Fig. 2. Wheellike Structure of a Register File with Four Multiple Overlapping Register Windows

stack cannot grow very large without overwriting itself. For this reason the register file is refined to emulate a push-down stack of register windows.

In figure 3 a structure to implement the push-down stack of register windows is shown. In this figure the register file has two windows in use. As more procedures and functions are called the unused register windows will be activated, but eventually the register file will become full. A full register file in this example would be W0, W1, and W2 in use, or (Next Window + 1) mod 4 = Save Window in this example. This is termed "full" because if W3 was used, the input parameters for procedure P0 in W0 could be altered. If the register file is full and another procedure or function call occurs, the contents of W0 will be pushed onto the saved window stack, then the values of Save Window, Next Window and Current Window will all be incremented by 1 modulus 4 in this case. This action is a window overflow. The opposite action, a window underflow, can occur during a procedure or function return. An underflow occurs if the Current Window = Save Window and a return is encountered. To handle an underflow, a window is popped off of the Saved Window Stack and is restored into the unused window adjacent to the Save Window (Save Window −1) mod 4 in this example. Then the Save Window, Current Window, and Next Window are all decremented by 1 modulus 4. The register file wheel can be viewed as rolling counterclockwise for a CALL/SAVE and clockwise for a RETURN/RESTORE.

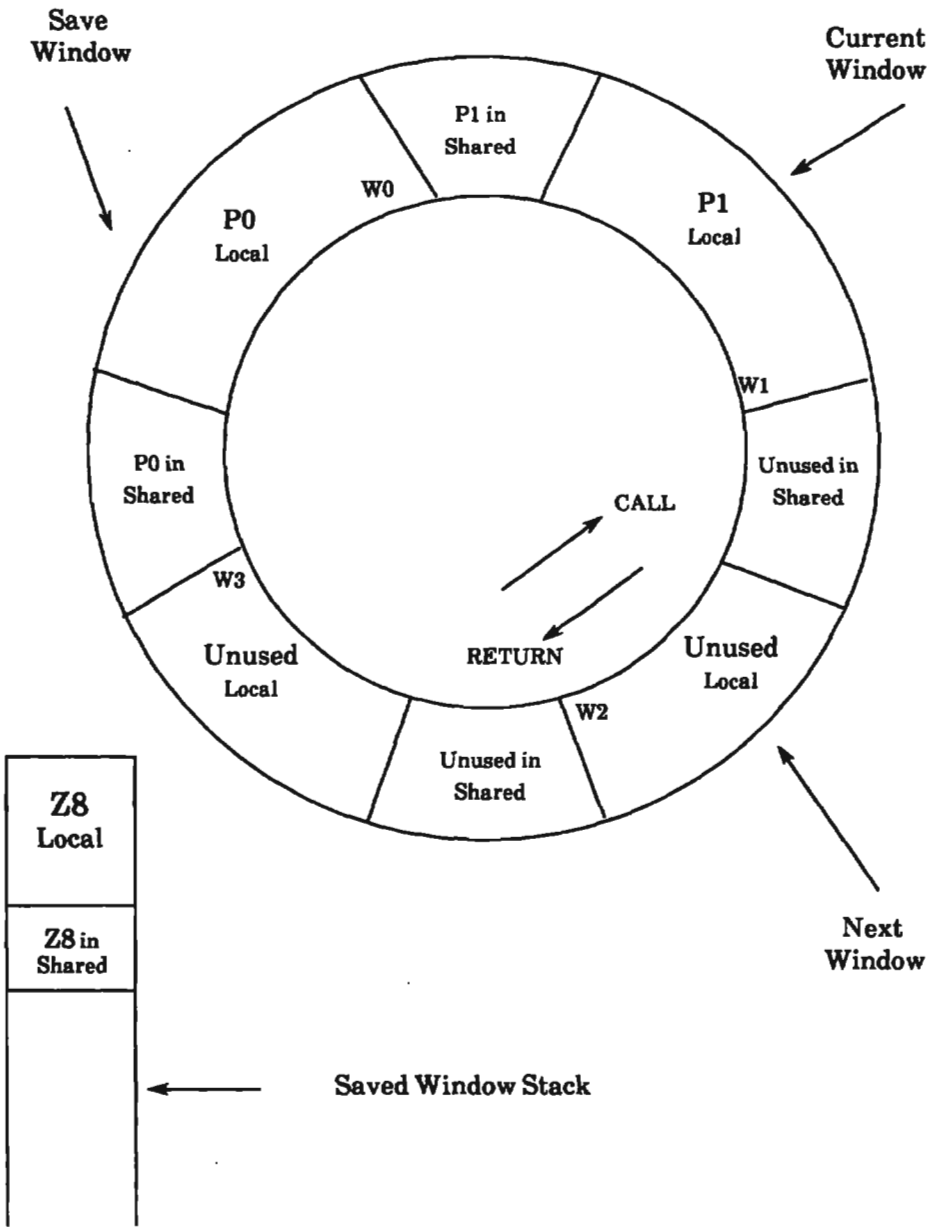


Fig. 3. Register File with Unused Register Windows [10]

The next issues concerning this type of register file organization is the number of register windows to provide, the number of local registers in each window and their organization, and finally how to optimize the management of this register file. A great deal of research has been done in this area as part of the Berkeley RISC II development [6, 4, 11]. In the Berkeley RISC research, the CALL/RETURN patterns, parameter passing, and scalar variable usage were analyzed within programs representing a wide variety of

processing needs. From their own research and the analysis of research done by others, the researchers concluded the following:

> In all cases, we saw that programs are organized in procedures and that procedure calls are frequent and costly in terms of execution time. Procedures usually have a few arguments and local variables, most of which are scalars and are heavily used. The (CALL/RETURN) nesting depth fluctuates within narrow ranges for long periods of time. [4]

This conclusion indicates that the register file with MORWs is a viable architectural technique. Since the nesting depth can be characterized as fluctuating within narrow ranges most of the time, a register file with enough windows to contain the average long-term fluctuation could avoid costly saves and restores to/from the saved window stack. Overlapping register windows are beneficial to computer architectures, due to the local variable and parameter usage patterns of a typical procedure. Finally, due to the frequency of procedure calls in most programs, the architecture will benefit from a windowed register file that is properly tuned for procedure calls and returns.

The Berkeley RISC researchers also studied strategies for managing this type of register file [11]. The main variables they were concerned with were the number of register windows to be pushed on or popped off the Saved Window Stack when an overflow or underflow occurred and the number of register windows that the register file should contain. Their findings can be summarized as follows.

The optimal management strategy would be a dynamic strategy that could predict the future behavior of the program being executed and from this information select the proper number of register windows to be pushed or popped. The researchers' attempts to do this, relying on the past behavior of the program, did not succeed. Also analyzed were static strategies. These strategies always pushed and popped a certain number of register windows off of the Saved Window Stack. The findings with these strategies were that pushing or popping only one register window from or to the Saved Window Stack seemed to work well, within a factor of two of the cost of the theoretical optimal strategy. For larger register files (more than eight register windows), pushing or popping two register windows at a time improved the register file performance slightly.

In terms of hardware, this type of register file is slightly expensive. Extra hardware must be included to detect overflows and underflows and to keep track of the current register window. The register windows also add an extra level of address decoding by requiring the register address to be added to the current window displacement to get the actual address within the register file. The use of fixed sized windows may cause utilization problems by not using all registers in a register window. In a large register file this may add up to a large amount of expensive wasted storage. The benefits are that this type of register file is regular in structure and fairly straightforward.

There are alternatives to this design, but there are trade-offs associated with these as well. Variable length register windows take care of the potential wasted storage problem. Because of this, a smaller register file may be possible. The problem is that detecting overflow and underflow becomes much more difficult since there are no fixed window boundaries. A third type of windowed register file, termed a Dribble-Back Register File, eliminates the problem of underflow and overflow detection. This type of register file takes care of saving and restoring register windows in the background during unused memory cycles. This method is very good for register files with small numbers of register windows since the overhead of saving and restoring windows is diminished. One trade-off with this method is that it can require a large memory bandwidth if it must accomplish a large number of save and restores. Another drawback is that if calls and returns occur in rapid succession, the program may have to wait while the register windows are being saved and restored [4].

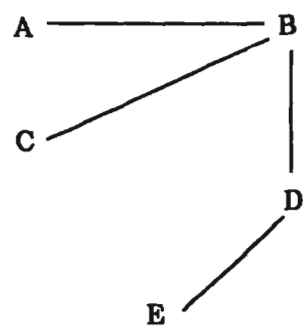
In the software arena, the windowed register file is very beneficial. The windows provide procedure calls that have very little overhead if the register file is properly tuned. This type of register file is also easier on the compiler writers in that the compiler does not have to manage the allocation of registers across procedure boundaries. The problem with this arrangement is that the hardware restricts a procedure to a single window of fixed size. This may limit some procedures that require an extreme amount of registers, but since the architecture should be fairly well tuned it should not be that much of a problem under normal circumstances.

COMPILER MANAGEMENT OF LARGE REGISTER FILES

A different method for dealing with a large register file is to allow the system's compiler to take care of the problem. In this method every procedure has access to all registers in the register file and it is up to the compiler to allocate, save, or restore each register as needed. A method for accomplishing this was developed by IBM for the IBM 801 project [7]. The technique likens register allocation to the graph coloring problem where the nodes of a graph must each be assigned a color so that no two nodes connected by a vertex are the same color.

Symbolic Registers	Instructions Using Register
A	1–5
B	3–8
C	6–7
D	8–9
E	9–10

Symbolic Registers Required to Coexist Connected



Coloring with Three Physical Registers

A and C	R0
B and E	R1
D	R2

Fig. 4. Coloring Example with Three Physical Registers

In the IBM approach the compiler generates an intermediate code in which every variable is allocated its own symbolic register. The compiler then compares the lifetimes required for each symbolic register. A graph is constructed where every symbolic register is a node in the graph and nodes are connected by vertices if there is a point in the program where the nodes (symbolic registers) must coexist. The compiler then colors the graph with the available physical registers so that no two symbolic registers connected by a vertex use the same physical register. The object in coloring is to minimize the number of

symbolic registers that share each physical register while maximizing the number of physical registers in use. The final result is the compiled code with the proper register saves and restores inserted to facilitate the sharing of the register file.

An example of coloring is given in figure 4. This example shows five symbolic registers, A through E, that are to be mapped onto three physical registers. The graph is drawn with each symbolic register as a node and a vertex connecting those nodes that must coexist within the processor's physical registers. Information regarding the instructions that use each symbolic register is used to determine those registers that must coexist. After the graph is constructed, it is easily determined that coloring A and C with R0, B and E with R1, and D with R2 allows the five symbolic registers to be mapped onto the three physical registers without conflict. If this example was limited to two physical registers, then coloring A, C, and D with R0 as well as B and E with R1 could be used.

This type of register file management greatly simplifies the circuitry required to deal with the register file. Basically no special circuitry other than the register file itself is required. This frees space within the processor to be used for other purposes, even more registers if desired. This method influences some design decisions. The design greatly benefits by providing the most registers possible. The compiler's job of register allocation becomes easier as more registers are provided. If too few registers are provided, registers may have to be saved and restored often and memory bandwidth may become a concern.

Requiring the compiler to take on the management responsibilities of the register file greatly increases its complexity. The compiler must optimize register file usage on top of everything else that must be done. A benefit of this technique is that all the processor's registers are available if they are required.

CURRENT RISC ARCHITECTURES

Berkeley RISC II is a 32-bit single chip processor built at the University of California at Berkeley. This processor is characterized by a small powerful instruction set and a large register file. The instruction set contains 39 instructions with only two instruction formats. The register file contains 138 registers and is implemented using eight overlapping register windows. Each register window contains six input registers (10–15), six output registers (26–31), and ten local registers (16–25). The register file also contains ten global registers (0–9) accessible to all register windows. Another characteristic is a delayed jump that always executes the instruction following a branch or jump to allow the smooth operation of the processor's instruction pipeline [4, 6].

The RISC processor produced by Sun Microsystems is called Scalable Processor ARChitecture (SPARC). The 32-bit SPARC processor is very similar to the Berkeley RISC II design. The register file is fairly large with 120 registers in its current implementation. The register file uses seven overlapping register windows. Each window contains eight input, eight output, eight local, and eight global registers. The instruction set, with three basic instruction formats, contains 46 integer operations and provides support for floating point and co-processor operations. This processor also uses delayed branching to simplify the instruction pipeline [8,9].

The IBM 801, named for the project's building number, is a RISC version of the IBM 370 architecture. This design utilizes compiler management of a 32-element register file. Backing up the register file is a very efficient cache system. The instruction set is a streamlined yet enhanced version of the IBM 370 instruction set and provides a delayed branch as an instruction separate from a normal branch. The system is heavily compiler dependent as the assembly language is not meant for normal programming use.

Stanford's Microprocessor without Interlocked Pipeline Stages (MIPS) uses only 55 instructions and 16 global registers. This system uses the compiler based register management methods employed in the IBM 801 system. In most computers the instruction pipeline hardware enforces pipe stage interlocking. Interlocking handles the case wherein two instructions in the instruction pipeline use the same registers as operands by enforcing some form of wait state between the instructions. The MIPS processor does away with the hardware technique and forces the compiler to enforce pipeline stage interlocking. To accomplish this task the compiler must order the instructions so that a pipline wait is never needed.

These four architectures differ in the hardware and software trade-offs made by their designers yet are still very similar. All four are register to register architectures, meaning that all instructions except LOADs and STOREs use only registers or constants as their operands. The instruction sets are all relatively small yet contain a powerful set of primitive instructions. Each processor performs single-cycle instruction execution due to streamlined control circuitry. Most of all, these processors were designed with this basic assumption – the more registers the better.

CONCLUSION

The IBM and MIPS usage of the register file as a global array of registers allows for maximum utilization of the register resources. But the cost paid in compiler complexity may be detrimental. Giving the compiler more responsibility in providing efficient use of the hardware may cause this utility to become an unwieldy, unmanageable beast.

Using a register file with Multiple Overlapping Register Windows increases the complexity of the register file circuitry yet yields many benefits. A study carried out at Carnegie-Mellon University [1] suggests that this type of register file management yields a substantial increase in processor performance. The Overlapping Register Windows free the compiler from globally optimizing register allocation yet provide procedures with a reasonable number of registers.

Computer designers are faced with many design trade-offs. In RISC architecture design, the management of the usually large number of registers becomes important because the performance of the architecture is directly correlated with how well the register file management performs. The Register File will always be a feature of RISC architectures, but a Register File with Multiple Overlapping Register Windows may become a characteristic of successful RISC architectures.

GLOSSARY

Berkeley RISC I and II – RISC architecture processors from the University of California at Berkeley (1980–1983).

CISC – Complex Instruction Set Computer.

Coloring – Graph coloring based algorithm for performing register allocation.

Compiler – A program that translates a High Level Language into another Language.

Cost – A measure of how well a management strategy performs.

Dribble-Back Register File – A windowed register file which performs the saving and restoring of register windows in the background during unused memory cycles.

IBM 801 – RISC version of an IBM 370 (1980).

MIPS – Stanford's RISC Microprocessor without Interlocked Pipeline stages (1983).

MORW – Multiple Overlapping Register Windows

Register File – A group of internal processor registers accessed with an address.

RISC – Reduced Instruction Set Computer.

Scalability – Refers to the ease in which an architecture can be reimplemented using the smaller feature sizes possible with VLSI technology advances.

SPARC – Sun Microsystems' Scalable Processor Architecture, RISC based processor implementation (1987).

VLSI – Very Large Scale Integrated Circuits.

STATUTORILY EXEMPT

REFERENCES

[1] Colwell, Robert P., Charles Y. Hitchcock III, Douglas E. Jensen, H.M. Brinkley Sprunt, and Charles P. Kollar. "Computer, Complexity, and Controversy." *IEEE Computer*, September 1985, pp. 8–19.

[2] Ditzel, David A. "Register Allocation for Free: The C Machine Stack Cache." Proceedings of the Symposium for Programming Languages and Operating Systems, Palo Alto, California, 1982, in *ACM SIGARCH Computer Architecture News*, Vol. 10, No. 2, March 1982, pp. 48–56.

[3] Hennesy, John, et al. "Hardware/Software Trade-offs for Increased Performance." Proceedings of the Symposium for Programming Languages and Operating Systems, Palo Alto, California, 1982, in *ACM SIGARCH Computer Architecture News*, Vol. 10, No. 2, March 1982, pp. 2–11.

[4] Katevenis, M. *Reduced Instruction Set Computer Architectures for VLSI.* Ph.D. Dissertation, Computer Science Department, University of California at Berkeley, October, 1983. Reprinted by MIT Press, Cambridge, Massachusetts, 1985.

[5] Markoff, John. "RISC Chips." *Byte*, November 1984, pp. 191–206.

[6] Patterson, David A. "Reduced Instruction Set Computers," *Communications of the ACM*, Vol. 28, No. 1, January 1985, pp. 8–21.

[7] Radin, George. "The 801 Minicomputer." Proceedings of the Symposium for Programming Languages and Operating Systems, Palo Alto, California, 1982, in *ACM SIGARCH Computer Architecture News*, Vol. 10, No. 2, March 1982, pp. 39–47.

[8] Sun Microsystems, Inc. A RISC Tutorial, 1987.

[9] Sun Microsystems, Inc. *The SPARC Architecture Manual*, 1987.

[10] Stallings, W. *Computer Organization and Architecture.* New York: Macmillan, 1987.

[11] Tamir, Yuval and Carlo H. Sequin. "Strategies for Managing the Register File in RISC." *IEEE Transactions on Computers*, Vol. C-32, No. 11, November 1983.

[12] Wallich, Paul. "Toward Simpler, Faster Computers." *IEEE Spectrum*, August 1985, pp. 38–45.