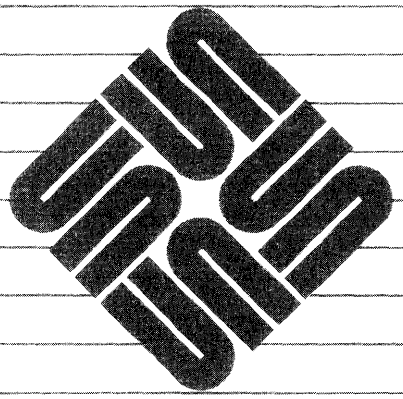
sun®
microsystems

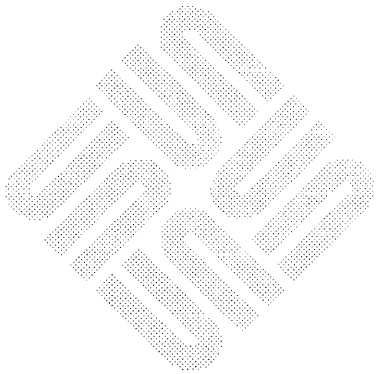
A RISC Tutorial



Part Number: 800-1795-10
Revision A of 9 May, 1988

Contents

Figures



1

Introduction

1.1. Scalable Processor Architecture

Sun Microsystems has designed a RISC architecture, called SPARC™, and has implemented that architecture with the Sun-4™ family of supercomputing workstations and servers. SPARC stands for Scalable Processor ARChitecture, emphasizing its applicability to large as well as small machines. SPARC systems have an open computer architecture — the design specification is published, and other vendors are producing microprocessors implementing the design. As with the Network File System (NFS™), we hope that the intelligent and aggressive nature of the SPARC design will become an industry standard.

The term "scalable" refers to the size of the smallest lines on a chip. As lines become smaller, chips get faster. However, some chip designs do not shrink well — they do not scale properly — because the architecture is too complicated. Because of its simplicity, SPARC scales well. Consequently, SPARC systems will get faster as better chip-making techniques are perfected.

Although this document is neither detailed nor highly technical, it assumes that you are acquainted with the vocabulary of a computer architecture. (An *architecture* is an abstract structure with a fixed set of machine instructions.) The first chapter answers the questions: what is RISC, and why is it useful? The second chapter gives an overview of the SPARC architecture. The third chapter compares the SPARC design with other RISC architectures, pinpointing the advantages of Sun's design.

1.2. What is RISC?

RISC, an acronym for Reduced Instruction Set Computer, is a style of computer architecture emphasizing simplicity and efficiency. RISC designs begin with a necessary and sufficient instruction set. Typically, a few simple operations account for almost all computations — these operations must execute rapidly. RISC is an outgrowth of a school of system design whose motto is "small is beautiful." This school follows Von Neumann's advice on instruction set design:

> The really decisive consideration in selecting an instruction set is *"simplicity of the equipment demanded by the [instruction set], and the clarity of its application to the actually important problems, together with [its speed] handling those problems."*

Simpler hardware, by itself, would seem of marginal benefit to the user. The advantage of a RISC architecture is the inherent speed of a simple design and the ease of implementing and debugging this simple design. Currently, RISC

machines are about two to five times faster than machines with comparable traditional architectures,† and are easier to implement, resulting in shorter design cycles.

RISC architecture can be thought of as a delayed reaction to the evolution from assembly language to high-level languages. Assembly language programs occasionally employ elaborate machine instructions, whereas high-level language compilers generally do not. For example, Sun's C compiler uses only about 30% of the available Motorola 68020 instructions. Studies show that approximately 80% of the computations for a typical program requires only about 20% of a processor's instruction set.

RISC is to hardware what the UNIX operating system is to software. The UNIX system proves that operating systems can be both simple and useful. Hardware studies suggest the same conclusion. As technology reduces the cost of processing and memory, overly complex instruction sets become a performance liability. The designers of RISC machines strive for hardware simplicity, with close cooperation between machine architecture and compiler design. At each step, computer architects must ask: to what extent does a feature improve or degrade performance and is it worth the cost of implementation? Each additional feature, no matter how useful it is in an isolated instance, makes all others perform more slowly by its mere presence.

The goal of RISC architecture is to maximize the effective speed of a design by performing infrequent functions in software, including in hardware only features that yield a net performance gain. Performance gains are measured by conducting detailed studies of large high-level language programs. RISC improves performance by providing the building blocks from which high-level functions can be synthesized without the overhead of general but complex instructions.

The UNIX system was simpler than other operating systems because its developers, Thompson and Ritchie, found that they could build a successful timesharing system with no records, special access methods or file types. Likewise, as a result of extensive studies, RISC architectures eliminate complicated instructions requiring microcode support, such as elaborate subroutine calls and text-editing functions. Just as UNIX retained the important hierarchical file system, recent RISC architectures retain floating-point functions because these functions are performed more efficiently in hardware than in software.

Portability is the real key to the commercial success of UNIX, and the same is true for RISC architectures. At the mere cost of recompilation, programs that run on VAXes or systems that use the various 68000 CPUs will run faster on RISC machines. RISC architectures are more portable than traditional architectures because they are easier to implement, thus permitting the rapid integration of new technologies as they become available. Users benefit because architectural portability allows more rapid improvements in the price/performance of computing.

† By comparable we mean architectures that cost about the same to implement. A Cray-2 supercomputer is not comparable to an IBM PC in this sense.

For computer architects, the word technology refers to how chips are made — how lines are drawn, how wide these lines are, and the chemical process involved. The use of gallium arsenide in fabrication, which creates faster chips, is an example of a recent development in chip technology.

1.3. RISC Architecture

The following characteristics are typical of RISC architectures. Although none of these are required for an architecture to be called RISC, this list does describe most current RISC architectures, including the SPARC design.

- Single-cycle execution. Most instructions are executed in a single machine cycle.
- Hardwired control with little or no microcode. Microcode adds a level of complexity and raises the number of cycles per instruction.
- Load/Store, register-to-register design. All computational instructions involve registers. Memory accesses are made with only load and store instructions.
- Simple fixed-format instructions with few addressing modes. All instructions are the same length (typically 32 bits) and have just a few ways to address memory.
- Pipelining. The instruction set design allows for the processing of several instructions at the same time.
- High-performance memory. RISC machines have at least 32 general-purpose registers and large cache memories.
- Migration of functions to software. Only those features that measurably improve performance are implemented in hardware. Software contains sequences of simple instructions for executing complex functions rather than complex instructions themselves, which improves system efficiency.
- More concurrency is visible to software. For example, branches take effect *after* execution of the following instruction, permitting a fetch of the next instruction during execution of the current instruction.

The real keys to enhanced performance are single-cycle execution and keeping the cycle time as short as possible. Many characteristics of RISC architectures, such as load/store and register-to-register design, facilitate single-cycle execution. Simple fixed-format instructions, on the other hand, permit shorter cycles by reducing decoding time.

Note that some of these features, particularly pipelining and high-performance memories, have been used in supercomputer designs for many years. The difference is that in RISC architectures these ideas are integrated into a processor with a simple instruction set and no microcode.

Moving functionality from run time to compile time also enhances performance — functions calculated at compile time do not require further calculating each time the program runs. Furthermore, optimizing compilers can rearrange pipelined instruction sequences and arrange register-to-register operations to reuse computational results.

1.4. Earlier Architectures

The IBM System/360, introduced in 1964, was the first computer to have an *architecture* (an abstract structure with a fixed set of machine instructions) separate from a hardware *implementation* (how computer designers actually built that structure). The IBM 360 architecture is still used today; IBM has brought out many computers implementing this architecture (or extensions of it) in various ways. Sometimes instructions are performed in hardware; other times in microcode.

Microcode is composed of low-level hardware instructions that implement higher-level instructions required by an architecture. At first, microcode was programmed by an elite group of engineers and then burned into ROM (read-only memory) where it could only be changed by replacing the ROM. In the early 1970s ROM was already quite dense — 8192 bits of ROM took up the same space as 8 bits of register.

The biggest problem was that the microcode was never bug-free and replacing ROMs became prohibitively expensive. So microcode was placed in read-write memory chips called control-store RAMs (random-access memory). In the mid 1970s, RAM chips offered a good solution, because RAM was faster, although more expensive, than the ferrite-core memory used in many computers. Thus, microcode ran faster than programs loaded into core memory.

Given the slow speed and small size of ferrite-core memory, complicated instruction sets were the best solutions for reducing program size and therefore, increasing program efficiency. Instruction set design placed great emphasis on increasing the functionality and reducing the size of instructions. Almost all computer designers believed that rich instruction sets would simplify compiler design, help alleviate the software crisis, and improve the quality of computer architectures.

Because of the scarcity of programmers and the intractability of assembly language, software costs in the 1960s rose as quickly as hardware costs dropped. This led the trade press to make dire predictions of an impending software crisis. The software crisis was somewhat diminished in the commercial sector by the development of high-level languages, the packaging of standard software products, and increases in CPU speed and memory size that allowed programmers to use medium-level languages. Clearly, complicated instruction sets did nothing to alleviate the software crisis.

The improvement of the integrated circuit in the 1970s made microcode memory even cheaper and faster, encouraging the growth of microprograms. The IBM 370/168 and the VAX 11/780 each have more than 400,000 bits of microcode. Because microcode permitted machines to do more, enhanced functionality became a selling point.

However, not all computer designers held these opinions. Seymour Cray, for one, believed that complexity was bad, and continued to build the fastest computers in the world by using simple, register-oriented instruction sets. The CDC 6600 and the Cray-1 supercomputer were the precursors of modern RISC architectures. In 1975 Cray made the following remarks about his computer designs:

> *"[Registers] made the instructions very simple... That is somewhat unique. Most machines have rather elaborate instruction sets*

involving many more memory references in the instructions than the machines I have designed. Simplicity, I guess, is a way of saying it. I am all for simplicity. If it's very complicated, I can't understand it."

Many computer designers of the late 1970s did not grasp the implications of various technological changes. At that time, semiconductor memory began to replace ferrite-core memory; integrated circuits were becoming cheaper and performing 10 times faster than core memory. Also, the invention of cache memories substantially improved the speed of non-microcoded programs. Finally, compiler technology had progressed rapidly; optimizing compilers generated code that used only a small subset of most instruction sets. All of this meant that architectural assumptions made earlier in the decade were no longer valid.

A new set of simplified design criteria emerged:

- Instructions should be simple unless there is a good reason for complexity. To be worthwhile, a new instruction that increases cycle time by 10% must reduce the total number of cycles executed by at least 10%.
- Microcode is generally no faster than sequences of hardwired instructions. Moving software into microcode does not make it better, it just makes it harder to modify.
- Fixed-format instructions and pipelined execution are more important than program size. As memory gets cheaper and faster, the space/time tradeoff resolves in favor of time — reducing space no longer decreases time.
- Compiler technology should simplify instructions, rather than generate more complex instructions. Instead of substituting a complicated microcoded instruction for several simple instructions, which compilers did in the 1970s, optimizing compilers can form sequences of simple, fast instructions out of complex high-level code. Operands can be kept in registers to increase speed even further.

1.5. Early RISC Machines

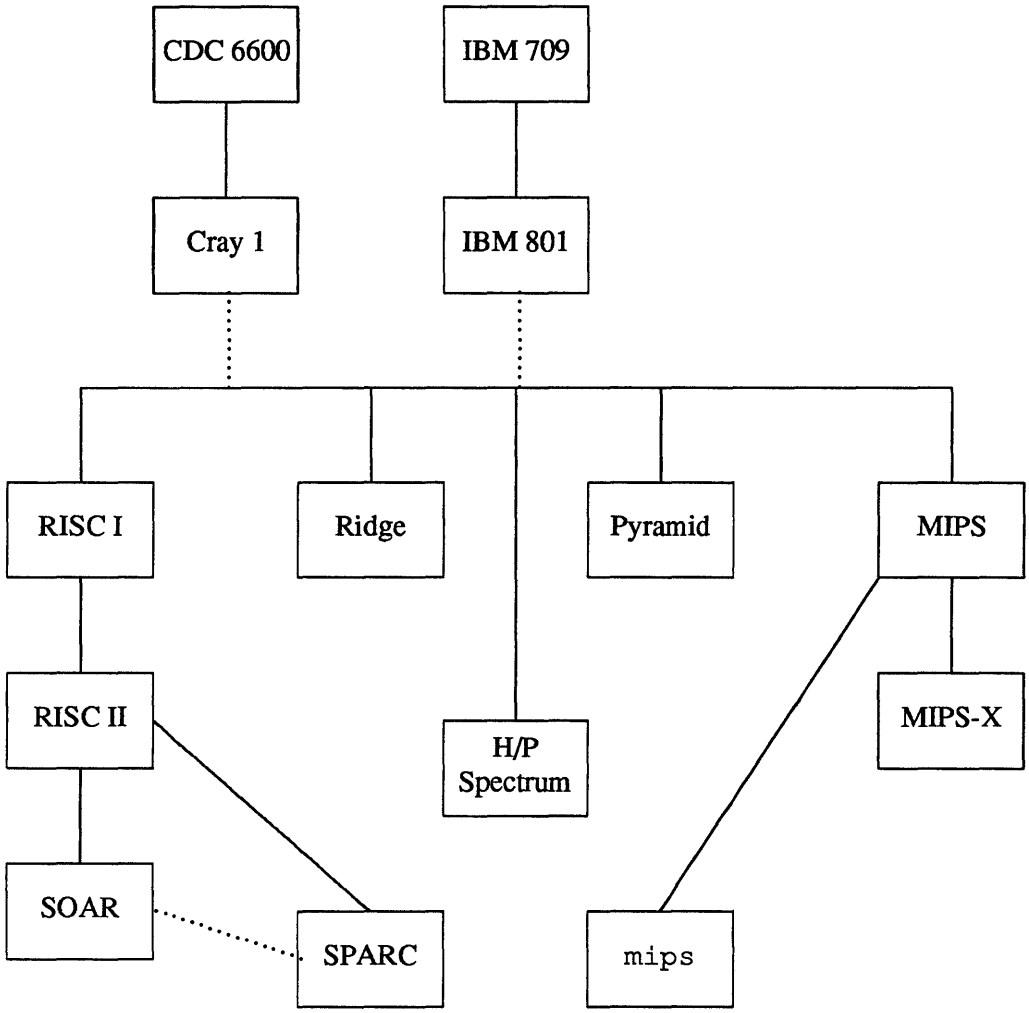
In the mid 1970s, some computer architects observed that even complex computers execute mostly simple instructions. This observation led to work on the IBM 801 — the first intentional RISC machine (although the term RISC had yet to be coined). Built from off-the-shelf ECL (emitter-coupled logic) and completed in 1979, the IBM 801 was a 32-bit minicomputer with simple single-cycle instructions, 32 registers, separate cache memories for instructions and data, and delayed branch instructions. The 801 was the predecessor of the chip now used as the CPU for the IBM PC/RT™, introduced early in 1986.

The term RISC was coined as part of David Patterson's 1980 course in microprocessor design at the University of California at Berkeley. The RISC-I chip design was completed in 1982, and the RISC-II chip design was completed in 1984. The RISC-II was a 32-bit microprocessor with 138 registers, and a 330-ns cycle time (for the 3-micron version). Even then, without the aid of elaborate compiler technology, the RISC-II outperformed the VAX 11/780 at integer arithmetic.

The MIPS project began at Stanford a short time later, with a group under the direction of John Hennessy. Hennessy's group declared that the acronym MIPS stood for Microprocessor without Interlocked Pipeline Stages, however, it is also a pun for Millions of Instructions Per Second. This group entrusted the compiler with pipeline management. The main goal of their design was high performance, perhaps at the expense of simplicity. The Stanford MIPS was a 32-bit microprocessor with 16 registers, and a 500-ns cycle time. The commercial processor marketed by the **mips** computer company is an outgrowth of the Stanford MIPS architecture.

Several other companies have announced RISC-type machines, including Ridge, Pyramid, and Hewlett-Packard with the Precision Architecture (Spectrum). The figure below shows how these various RISC architectures are related.

Figure 1-1 *Genealogy of RISC Architectures*



1.6. RISC's Speed Advantage

Using any given benchmark, the performance, P, of a particular computer is inversely proportional to the product of the benchmark's instruction count, I, the average number of clock cycles per instruction, C, and the inverse of the clock speed, S:

$$P = \frac{1}{I \cdot C \cdot \frac{1}{S}}$$

Let's assume that a RISC machine runs at the same clock speed as a corresponding traditional machine; S is identical. The number of clock cycles per instruction, C, is around 1.3 to 1.7 for RISC machines, but between 4 and 10 for traditional machines. This would make the instruction execution rate of RISC machines about 3 to 6 times faster than traditional machines. But, because traditional machines have more powerful instructions, RISC machines must execute more instructions for the same program, typically about 20% to 40% more. Since RISC machines execute 20% to 40% more instructions 3 to 6 times more quickly, they are about 2 to 5 times faster than traditional machines for executing typical large programs.

Compiled programs on RISC machines are larger than compiled programs on traditional machines, partly because several simple instructions replace one complex instruction and partly because of decreased code density. All RISC instructions are 32 bits wide, whereas some instructions on traditional machines are narrower. But the number of instructions actually executed may not be as great as the increased program size would indicate. Global registers, for example, often simplify call/return sequences so that context switches become less expensive.

Designers of RISC machines dramatically reduce the clock cycles per instruction while slightly increasing the instruction count per program, resulting in an overall performance increase. Moreover, RISC architectures scale better to new technology than more complicated architectures. Sometimes architectural cleverness backfires — because of complicated design, the performance of a machine will not improve at the same rate as technology advances. Simple RISC architectures, by contrast, will scale upwards as cycle times decrease and memory sizes increase.

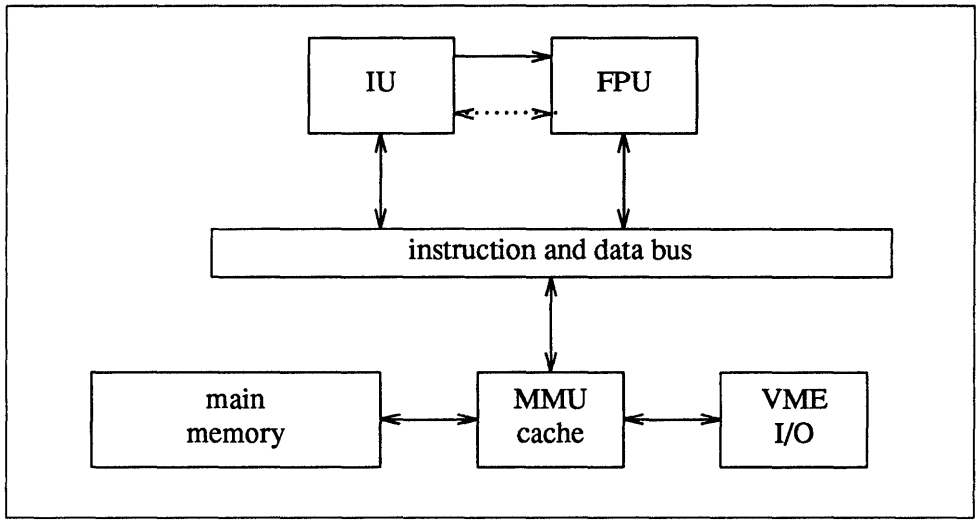
2

SPARC Architecture

An architecture, or abstract design, often spans several hardware implementations. This chapter introduces the SPARC architecture, without going into specifics about particular implementations.

The SPARC CPU is composed of an Integer Unit (IU) that performs basic processing and a Floating-Point Unit (FPU) that performs floating-point calculations. According to the architecture, the IU and the FPU may or may not be implemented on the same chip. Although not a formal part of the architecture, SPARC-based computers from Sun Microsystems have a memory management unit (MMU), a large virtual-address cache for instructions and data, and are organized around a 32-bit data and instruction bus.

Figure 2-1 *Sample SPARC Implementation*



The integer and floating-point units operate concurrently. The IU extracts floating-point operations from the instruction stream and places them in a queue for the FPU. The FPU performs floating-point calculations with a set number of floating-point arithmetic units (the number is implementation-dependent). The SPARC architecture also specifies an interface for the connection of an additional coprocessor.

2.1. Instruction Categories

The SPARC architecture has about 50 integer instructions, a few more than earlier RISC designs, but less than half the number of Motorola 68000 integer instructions. SPARC instructions fall into five basic categories:

1. Load and store instructions (the only way to access memory). These instructions use two registers or a register and a constant to calculate the memory address involved. Half-word accesses must be aligned on 2-byte boundaries, word accesses on 4-byte boundaries, and double-word accesses on 8-byte boundaries. These alignment restrictions greatly speed up memory access.
2. Arithmetic/logical/shift instructions. These instructions compute a result that is a function of two source operands and then place the result in a register. They perform arithmetic, tagged arithmetic, logical, or shift operations. Tagged instructions are useful for implementing artificial intelligence languages such as LISP, because tags provide interpreters with the type of arithmetic operands.
3. Coprocessor operations. These include floating-point calculations, operations on floating-point registers, and instructions involving the optional coprocessor. Floating-point operations execute concurrently with IU instructions and with other floating-point operations when necessary. This architectural concurrency hides floating-point operations from the applications programmer.
4. Control-transfer instructions. These include jumps, calls, traps, and branches. Control transfers are usually delayed until after execution of the next instruction, so that the pipeline is not emptied every time a control transfer occurs. Thus, compilers can be optimized for delayed branching.
5. Read/write control register instructions. These include instructions to read and write the contents of various control registers. Generally the source or destination is implied by the instruction.

2.2. Register Windows

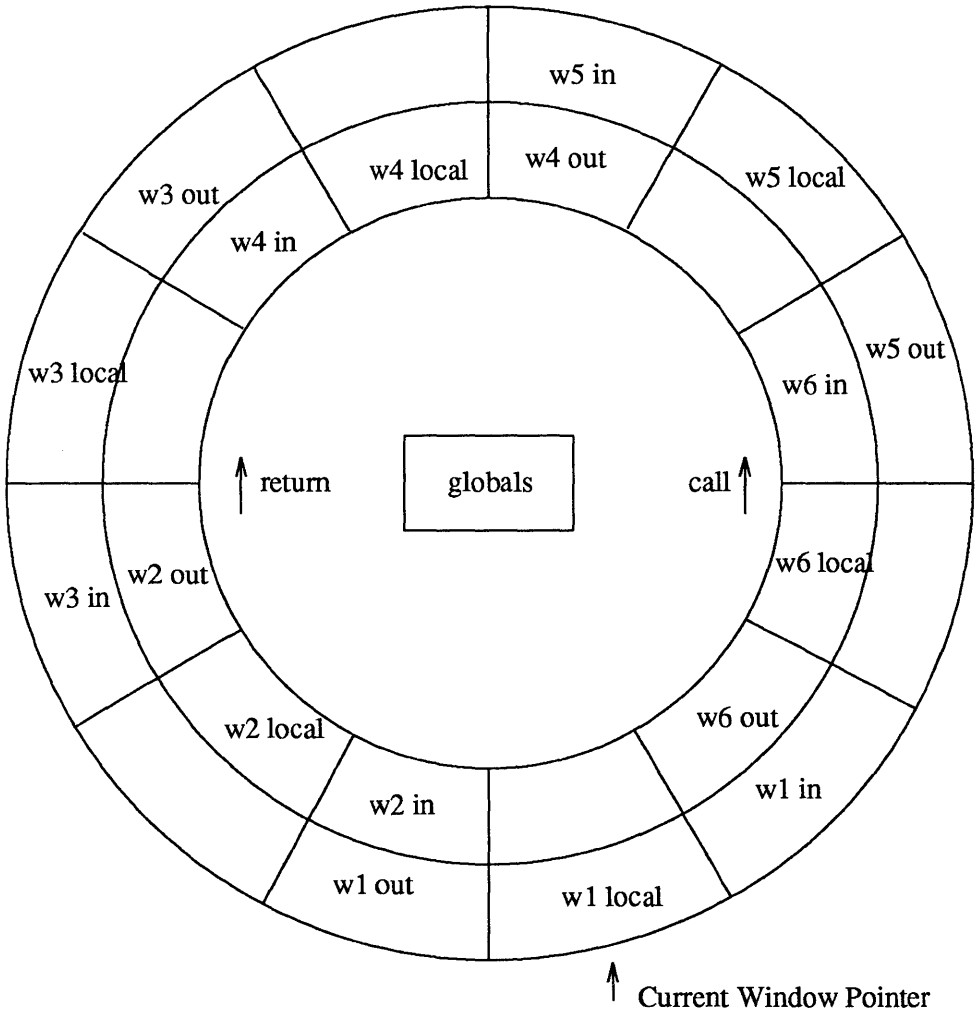
A unique feature contributing to the high performance of the SPARC design is its overlapping register windows. An analogy can be made comparing the register windows with a rotating, high-performance tire. Some part of the tire's tread is always on the ground. As it rotates, the tire's zigzag tread grips a different portion of the road. The zigzag tread is analogous to the overlap of register windows. Results left in registers become operands for the next operation, obviating the need for extra load and store instructions.

According to the architectural specification, there may be anywhere between 6 and 32 register windows, each window having 24 working registers, plus 8 global registers.† Each register window is logically divided into three groups: 8 `in` registers, 8 `local` registers, and 8 `out` registers. The `out` registers for one window become *in* registers for the next; they are, in fact, the same registers. The current window pointer keeps track of which window is currently active. The figure below is a diagram of a SPARC implementation with 6 register

† The first implementation has 7 register windows with 24 registers each (but count only 16 since 8 overlap), plus 8 global registers, for a total of 120 registers.

windows. Note that the first actual SPARC implementation has 7 windows, so in addition to the windows in this diagram, there would also be `w0 in`, `w0 local`, and `w0 out`.

Figure 2-2 *Overlapping Register Windows*



For a function call, the register windows rotate counterclockwise; for a return from a function call, they rotate clockwise.

The alternative to register windows encompasses slower, more elaborate register allocations, which must be performed during compile time. For languages such as C, Pascal, and Modula-2, this strategy is merely time consuming. For exploratory programming environments such as Lisp and Smalltalk, where compiler speed is crucial to improving programmer productivity, users may find slow optimizing compilers unacceptable, and unable to achieve the potential performance available on SPARC machines.

Recent research suggests that register windows and tagged arithmetic, found in SPARC systems but not in other commercial RISC machines, are sufficient to provide excellent performance for expert system development requiring AI

languages such as Lisp and Smalltalk.† Recent benchmark experience with SPARC systems supports earlier evidence.

2.3. Traps and Exceptions

The SPARC design supports a full set of traps and interrupts. They are handled by a table that supports 128 hardware and 128 software traps. Even though floating-point instructions can execute concurrently with integer instructions, floating-point traps are precise because the FPU supplies (from the table) the address of the instruction that failed.

2.4. Memory Protection

Some SPARC instructions are privileged and can only be executed while the processor is in supervisor mode. This instruction execution protection ensures that user programs cannot accidentally alter the state of the machine with respect to its peripherals and vice versa.

The SPARC design also provides memory protection, which is essential for smooth multitasking operation. Memory protection makes it impossible for user programs that have run amok to trash the system, other user programs, or themselves.

† D. Ungar, R. Blau, P. Foley, A.D. Samples, D. Patterson, "Architecture of SOAR: Smalltalk on a RISC," in *Proceedings of the 11th Annual International Symposium on Computer Architecture*, Ann Arbor, 1984.

3

An Open Architecture

3.1. Advantages of Open Architecture

The SPARC design is the first open RISC architecture, and one of the few open CPU architectures. An architectural standard would lift the industry out of often useless debates over the merits of various microprocessors. Standard products are more beneficial than proprietary ones, because standards allow users to acquire the most cost-effective hardware and software in a competitive multi-vendor marketplace. Integrated circuits would come from chip vendors, while software would be supplied by systems vendors. This advantage is lost when users are limited by a processor with proprietary hardware and software.

RISC architectures, and the SPARC design in particular, are easy to implement because they are relatively simple. Since they have short design cycles, RISC machines can absorb new technologies almost immediately, unlike complicated computer architectures.

The SPARC architecture is an aggressive, forward-thinking design. Even in the first implementation, processor cycle time is very fast — equivalent to the access time of static random-access memory (SRAM) rather than dynamic random-access memory (DRAM). Because registers are used intensively in a load/store architecture, the high cost of fast memory (as with SRAM) can be concentrated where it is used the most — in registers. Because the clock cycles per instruction are kept to a minimum, pipelining is simple and fast, since few restarts are necessary. So the high performance of SPARC systems results from both simple design and technological leverage.

3.2. SPARC Design and RISC

Like other RISC architectures, SPARC systems provide:

- Single-cycle execution. All instructions except loads, stores, and floating-point operations can be executed in one machine cycle.
- Simple instruction format. All instructions are 32 bits wide and word-aligned in memory. Op-codes and addresses are always in the same place, so decoding hardware can be simplified.
- Register-intensive architecture. Instructions operate on two registers or on a register and a constant, placing the result in a third register. The only way to access memory is with load and store instructions.
- Large register windows. The processor has access to a large number of registers configured into overlapping sets, so that compilers can automatically cache values and pass parameters in registers.

- Delayed control transfer. The processor fetches the next instruction following a control transfer before completing the transfer. Compilers can rearrange code, placing useful instructions after a delayed control transfer, thus maximizing throughput.

3.3. How SPARC Design is Different

SPARC systems were designed to support:

- the C programming language and the UNIX operating system,
- numerical applications (using FORTRAN), and
- artificial intelligence and expert system applications using Lisp and Prolog.

Supporting C is relatively easy; most modern hardware architectures are able to do so. The one essential feature is byte addressability. However, numerical applications require fast floating point and artificial intelligence applications require large address spaces and interchangeability of data types.

The floating-point processor, with pipelined floating-point operation capabilities, achieves the high performance needed for numerical applications. Floating-point coprocessors are generally not part of RISC machines, but they are available for microprocessors such as the Motorola 68020 and the Intel 80386, and for SPARC systems as well.

For artificial intelligence and expert system applications, SPARC systems offer tagged instructions and word alignment. Because languages such as Lisp and Prolog are often interpreted, word alignment makes it easier for interpreters to manipulate and interchange integers and different types of pointers. In the tagged instructions, the two low-order bits of an operand specify the type of operand. If an operand is an integer, most of the time it is added to (or subtracted from) a register. If an operand is a pointer, most of the time a memory reference is involved. Language interpreters can leave operands in the appropriate registers, greatly improving the performance of exploratory programming environments.

The SPARC architecture does not specify a memory management unit (MMU) because we expect the same processor to be used in different types of machines. For example, a single-user machine with embedded applications, such as the Macintosh, does not need an MMU. By contrast, a multitasking machine used for timesharing, such as a traditional UNIX box, needs a paging MMU. Furthermore, a multiprocessor such as a vector machine or hypercube requires specialized memory management facilities. The SPARC architecture can be implemented with a different MMU configuration for each of these purposes, without affecting user programs.

3.4. Speed Advantage of SPARC Systems

Recall the equation in the first chapter, where the performance, P, of a processor is inversely proportional to the product of a benchmark's instruction count, I, the average clock cycle per instruction, C, and the inverse of the clock speed, S:

$$P = \frac{1}{I \cdot C \cdot \frac{1}{S}}$$

Working this equation for SPARC systems and for two popular microprocessors,

we come up with these numbers (*I* indicates millions of instructions so *P* is in MIPS):

Figure 3-1 *Processor Performance*

Processor Performance				
cpu	*I*	*C*	*S*	*P*
Motorola 68030	1.0	5.2	16.67	3.21
Intel 80386	1.1	4.4	16.67	3.44
SPARC	1.2	1.3	16.67	10.69

Thus, SPARC systems have a considerable theoretical performance advantage over other microprocessors on the market. The table compares three processors running at the same clock speed; higher clock speeds are possible with all three processors.

3.5. SPARC Machines and Other RISC Machines

The SPARC design has more similarities to Berkeley's RISC-II architecture than to any other RISC architecture. Like the RISC-II architecture, it uses register windows in order to reduce the number of load/store instructions. The SPARC architecture allows 32 register windows, but the initial implementation has only 7 windows. The tagged instructions are derived from SOAR, the "Smalltalk On A RISC" processor developed at Berkeley after implementing RISC-II.

Until recently, RISC architectures have performed poorly on floating-point calculations. The IBM 801, for example, implemented floating-point operations in software. The Berkeley RISC-I and RISC-II outperformed a VAX 11/780 in integer arithmetic, but not in floating-point arithmetic. This was also true of the Stanford MIPS processor. SPARC systems, on the other hand, are designed for optimal floating-point performance, and support single-, double-, and extended-precision operands and operations, as specified by the ANSI/IEEE 754 floating-point standard.

High floating-point performance results from concurrency of the IU and FPU. The integer unit loads and stores floating-point operands, while the floating-point unit performs calculations. If an error (such as a floating-point exception) occurs, the floating-point unit specifies precisely where the trap took place; execution is expediently resumed at the discretion of the integer unit. Furthermore, the floating-point unit has an internal instruction queue; it can operate while the integer unit is processing unrelated functions.

3.6. Conclusion

SPARC systems deliver very high levels of performance. The flexibility of the architecture makes future systems capable of delivering performance many times greater than the performance of the initial implementation. Moreover, the openness of the architecture makes it possible to absorb technological advances almost as soon as they occur.