# A Comparison of RISC Architectures

**Evaluating the newest chips for your needs can take some time and thought. Here's help in deciding what's important to consider.**

*Richard S. Piepho*
*William S. Wu*

*AT&T Bell Laboratories*

Numerous new RISC architectures have appeared in the marketplace over the past few months. Among these are the Intel i860, the Motorola 88000, and the Sun Microsystems Sparc architectures. Each claims great performance increases over the existing CISC architectures and superior performance and features over their RISC rivals.

Here, we compare and analyze the relative strengths and weaknesses of the three architectures in a number of key architectural areas. Based on this comparison, we assess their advantages and disadvantages. We seek to determine whether one architecture is clearly superior or inferior in the long term because of sufficient advantages or disadvantages it possesses over the others.

First we discuss the relative importance of an architectural comparison, as opposed to a comparison of implementations, and follow it with a high-level overview of each of the architectures. We examine, in detail, each of the architectures on a number of key architectural areas. Finally, we summarize the overall relative strengths and weaknesses of each architecture. (We also explain some of the specialized architectural vocabulary in the accompanying Definition of Terms.)

## Architecture vs. implementation

Various proposals for drawing the line between computer architecture and computer implementation have existed since the term "computer architecture" was first used in the description of the IBM System/360. The original strict definition proposed by Blaauw[1] limited the architecture to just the instruction set and execution model. All else makes up the implementation. A more-encompassing definition proposed by Stone[2] sets the architecture as the "instruction set and structure down to the functional modules" of the system. Various other definitions fall between these two extremes.

For our purposes, however, we adopt the original strict definition. We define the architecture as only software-visible features—including the basic instruction set and memory management architectures. It does not include the specification of the functional modules used to implement these features.

0272-1732/89/0800-0051 © 1989 IEEE

# Definition of Terms

**ABI**, or Unix System V application binary interface for a CPU architecture, defines a "binary" system interface standard. This standard supports compiled application programs running on computer systems that are based on the same CPU architecture.

An **atomic** instruction retains exclusive use of a flag (for example, a semaphore) through completion of the instruction cycle. Exclusive use of a flag prevents the flag from being modified while the instruction operates on it.

Byte-ordering or addressing schemes called **big endian** and **little endian** set the format for sending data to a microcomputer. Big-endian format sends the most significant byte first, while little endian sends the least significant byte first. Figure A shows big-endian byte ordering for a 32-bit word; Figure B shows little-endian byte ordering for a 32-bit word.

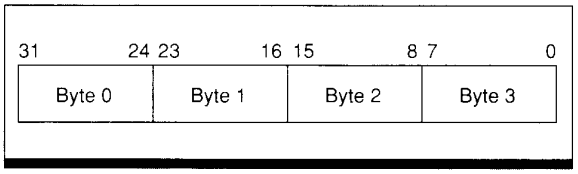


**Figure A. Big-endian ordering for a 32-bit word.**

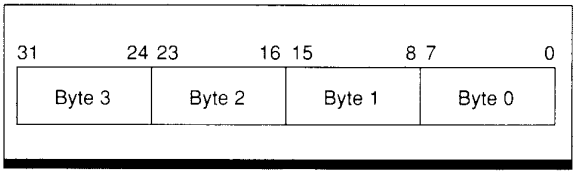


**Figure B. Little-endian ordering for a 32-bit word.**

A small, high-speed **cache** stores the most frequently used main memory locations. It typically requires only one to two processor cycles to access as compared with 10 to 20 cycles for main memory access. A cache usually can hold 1 Kbyte to 512 Kbytes of data.

The **cache coherency** protocol lets the hardware (or software) ensure that only one logically correct value exists for each program variable. In multiprocessing systems with each CPU containing a local cache, multiple copies of program variables can exist in the system. Each CPU can be attempting to modify and/or access its copy of the program variables simultaneously. The program variable copies could then become inconsistent (with each CPU seeing a different value of a program variable) without some hardware and/or software ensuring that some form of consistency or coherency is enforced.

**CISC** indicates a complex instruction set computer or computing.

A processor's **condition codes**, or information bits, allow the software programmer (and the hardware) to determine whether the result of a comparison (or other arithmetic operation) was positive, negative, or zero and whether it caused an overflow.

A graphics unit, for a given viewpoint, discards and does not display the nonvisible surfaces of objects in a scene through a process called **hidden-surface elimination.**

A **leaf procedure** will not call any other procedure.

The hardware unit or component called a **memory management unit,** or MMU, translates virtual addresses (those seen by software) into physical addresses (those seen by hardware). The hardware uses the memory-management translation information (page and segment table entries) to translate an address. It then stores the translation information in the TLB. (See TLB.) In addition, the hardware provides program isolation and protection (memory protection) by examining permission data in the translation information.

The **MESI** memory protocol ensures cache coherency between multiple write-back caches. Any given cache entry, depending on how it has been accessed, falls into one of four states: modified (M), exclusive (E), shared (S), or invalid (I).

A **page** is the smallest managed unit of a virtual memory scheme. The system maintains separate virtual-to-physical translation information (and, in some cases, protection information) for each page.

The **Phong-shading** graphics technique helps a graphics unit shade an object. The graphics unit linearly interpolates the normals at the vertices of a polygon along the edges. Then it interpolates the normals at the edges along a scan line. At each pixel along the scan line, the interpolated normal is used in the lighting model to determine the color at that pixel. For example, consider the triangle with four scan lines in Figure C.

Given the normals at vertices A, B, and C, the unit interpolates the normals along edges AB and AC. Then using the interpolated normals of the two edges at horizontal scan line 2, the unit interpolates the normals along scan line 2 between the edges to calculate the shade for each pixel. The unit then repeats the interpolation for scan lines 3 and 4.[3]

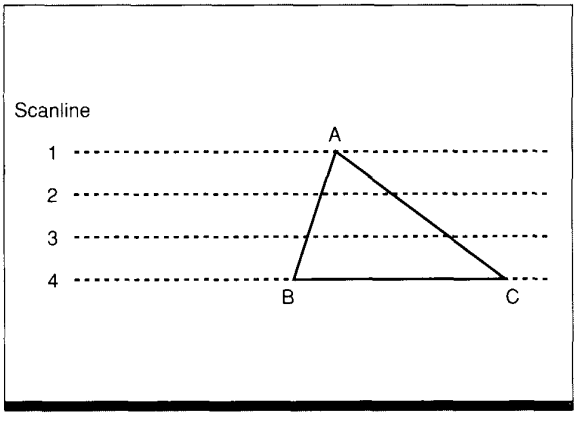


**Figure C. A 4-scanline figure.**

In a register organization scheme called **register windowing** a group of register banks (each with, for example, 32 registers) is arranged as a circular buffer. During execution, software is only "aware" of a single bank, or window, of registers. Each procedure call, however, results in a new window of registers being transparently allocated to the new procedure, thereby eliminating the need to save registers on each procedure call. Similarly, as each procedure completes and returns, the system readjusts the current window back to the correct window.

**RISC** indicates a reduced instruction set computer or computing.

A **semaphore** is a hardware/software flag that indicates the status of an activity. Typically, it signals whether or not a shared resource can be accessed. A **semaphore instruction** is a special atomic instruction for accessing the flag.

**Smalltalk** is a high-level, object-oriented programming language developed by Xerox PARC.

A **spin-on-the-lock** situation occurs when a program is in a loop constantly testing a semaphore to see if access to the related resource is allowed. Here, the semaphore functions like a lock.

An **SPL level** is a Unix interrupt level in a system that supports multiple levels of interrupts. A higher priority interrupt would logically be at a higher SPL level than a lower priority interrupt. Only those incoming interrupts at a higher SPL level than the current one actually cause an interrupt to be acknowledged by the processor. Raising or lowering the SPL level thereby increases or decreases the number of interrupts that the processor will acknowledge. Running at a high SPL level can essentially disable the processor from acknowledging interrupts.

**Tagged arithmetic** provides a primitive means of checking for consistency in data type thereby supporting the most frequent cases in the Smalltalk and Lisp languages. Many languages, including Smalltalk and Lisp, do not provide data type declarations. Therefore the type (pointer, data) of a program variable cannot be checked until the program is executed. Thus the operand type (and whether they match or not) must be checked before performing arithmetic instructions.

A **test-and-set** instruction typically tests a memory location (flag/semaphore) and updates it according to the flag's value. It is atomic in that after the flag is read and before the flag is updated, the CPU executing the instruction will not allow access to the flag.

A graphics unit transforms a displayed wireframe drawing into a **3D shaded** object by saving only the lines that form the surface of the wireframe's polygons and then filling in (shading) between these lines.

A **TLB,** or translation lookaside buffer, is the memory cache of the most recently used page table entries within the MMU.

A **TLB hit rate** is the cache hit rate achieved in the TLB. It reflects the percentage of memory accesses whose translation information (page table entry) is contained in the TLB.

**Virtual memory** is the memory space as understood by the software programmer. It allows each software application to "see" a uniform, large address space independent of the number of applications running on a system or the actual size of main memory of that system. The MMU maps, or translates, virtual memory into the actual (physical) memory of the system.

A dynamic, RAM-resident ***z*-buffer** supports graphics processing. It has a one-to-one correspondence with a frame buffer. That is, each pixel in the frame buffer has a corresponding location in the *z*-buffer. Each *z*-buffer location contains the depth (*z*) value (the pixel's distance from the viewer) of the object being displayed at the corresponding pixel in the frame buffer. Before drawing a pixel, the graphics unit compares the *z* value of the object, at that pixel, with the value in the *z*-buffer. The unit updates the pixel only if the object is closer to the viewer than that indicated currently in the frame buffer.

**Why architecture instead of implementation?** The RISC architectures offer large performance increases over currently available CISC architectures. As a result, almost every computer vendor evaluates the RISC architectures to verify performance claims and to determine which, if any, best fits its applications and strategic directions. Unlike past chip decisions (for example, whether to use an 8086 or an 68000) however, selecting a RISC chip and breaking user software compatibility with current product lines now becomes a major corporate decision. Software compatibility is increasingly important due to the rise of industry-standard application binary interfaces (ABIs). In addition, the number of available user application packages continues to grow. As a result, changes to an architecture become very difficult while changes to an implementation become increasingly easy.

Changing an architecture, in general, implies that changes will have to be made to user application software. Since most computer vendors do not write all of their own applications and because of the enormous number of packages that would have to be updated, the cost of such a change is very high. In some cases additions to an architecture could be made in such a manner that existing user application software is both forward- and backward-compatible. In general, however, this is not the case, and the selection of a good architecture is critical.

Changes to an implementation, however, imply that only changes to the hardware and possibly the operating system software will be necessary. Since vendors upgrade both the hardware and operating system on a regular basis (to include the latest chip implementation), the added cost of changes in an implementation remains small in comparison to the user software changes. Any limitations in a given implementation can be (and usually are) reduced or circumvented in the next implementation. Therefore the selection of a RISC based on a given implementation is not as critical.

Overall then, the more important selection criteria in selecting a RISC chip is its architecture, both the current and future implementations as opposed to just the architecture's current implementation.

**Architecture evaluation examples.** In the process of evaluating RISC architectures, we have seen numerous performance comparisons and architectural "evaluations" based on these comparisons. (Most have been conducted, it seems, by the companies selling one architecture or another.) While these evaluations have been extensive and have pointed out numerous potential shortcomings, most of these evaluations compare specific implementations of the architectures in question and not the architectures themselves. As a result, the shortcomings tend to be characteristic of the implementations and not the architectures. Table 1 lists some of the architectural shortcomings being put forth for each of the architectures.

**Table 1.**
**Claimed Architectural Shortcomings.**

| Architecture | Deficiency |
|---|---|
| i860 | No cache coherency for internal caches |
| 88000 | No dual-cache tags<br>Only supports MESI model of cache coherency |
| Sparc | Single address/data bus<br>No separate address adder |

In each of these cases, the proclaimed architectural shortcoming is, in fact, a feature of the implementation and not a feature of the architecture. The number of external buses, while a major component of the performance of RISC implementations, is not a feature of the architecture. The number, speed, and width of external buses can be (and is) changed from implementation to implementation without affecting the architecture. The support of cache coherency and the exact form of that support is, again, a crucial feature in the implementation of multiprocessor systems but is not a feature of the architecture. Cache coherency can be added, deleted, or changed without affecting the underlying processor architecture.

While many of the analyses being performed may have concentrated on specific implementations as opposed to the underlying architecture, we point out that the architectures are not without shortcomings nor all equal. On the contrary, the architectures, while on the surface quite similar, are quite different when examined in detail.

# Overview of architectures

The i860, 88000, and Sparc are labeled and marketed as RISC architectures. They all satisfy the key aspects of RISC design[3] and share some "prominent" RISC characteristics. These shared key characteristics are:

- single-cycle execution (for most instructions),
- simple load/store interface to memory,
- register-based execution,
- simple fixed-format and fixed-length instructions,
- simple addressing modes,
- large register set or register windows, and
- delayed branch instructions.

For some particular target markets, the vendors have also added sets of instructions that are not frequently used in general-purpose computing. For example, the

i860 provides a set of graphics and vector instructions, the 88000 offers an extensive set of bit-field instructions, and the Sparc includes instructions on tagged data. Probably, to some RISC purists/minimalists, the addition of such seemingly extraneous instruction sets disqualifies their classification as RISC architectures. However, in our opinion, the key point of RISC is the design philosophy of simplicity and efficiency. That is, RISC affords an efficient use of hardware resources via judicious simplification of the semantics of a processor's instruction set and encoding of the instruction set. These special instructions do not preclude the three architectures from being classified as RISC architectures.

To avoid a proliferation of memory management architectures, each of the architectures also includes a memory management definition.

# Architectural comparison

In examining the architectures of the i860, 88000, and Sparc, we look closely at the following areas:

- miscellaneous instructions,
- branches,
- memory operands and addressing modes,
- registers,
- data types and alignment,
- floating-point units, and
- memory management.

**Miscellaneous instructions.** In addition to the standard set of RISC instructions, each architecture includes fairly unique (at least for RISC architectures) instructions targeted for specific applications. The special i860 instructions support graphics processing as well as parallel operation of the integer and floating-point units. The graphics processing instructions include an extensive set of both pipelined and nonpipelined instructions, which support $z$-buffer operations, Phong shading, and pixel arithmetic. These capabilities provide superior support in graphics applications that perform hidden-surface elimination and 3D shading. However, since these instructions use the software-visible floating-point pipeline, their use is limited to libraries and specially coded routines. (We discuss this aspect further later.) For applications outside of the graphics area, these capabilities will not provide any measurable benefits.

The i860 also supports the parallel initiation of the integer and floating-point units via the dual-instruction-mode prefix. Use of this prefix causes the next two instructions to be initiated in parallel (assuming that one is an integer instruction and one is a floating-point instruction). For general-purpose applications, which typically perform few floating-point operations, the addition of such parallelism does not provide any significant benefit. Alternatively, for those applications that perform extensive floating-point operations, such parallelism provides a significant performance improvement. However, since the compiler must generate different code to take advantage of the parallelism (and the current compiler does not), it is unclear whether high-level-language programs will be able to make use of this capability. To the extent that an application's key routines and libraries can be written in assembly language, much of the performance improvement can be achieved.

The unique 88000 instructions are an extensive set of bit-field instructions. They provide the capability to set/clear and extract/insert values into bit fields of variable length and position. (Further discussion appears later.)

The unique Sparc instructions support tagged arithmetic. They provide the capability to tag data and pointers differently so that detection of illegal operations on the data or pointers can be detected. (We discuss this further later.)

*Semaphores.* The three architectures support some kind of semaphore or atomic test-and-set type of instruction. Semaphore instructions are an increasingly important part of the architecture due to the increase in the number of shared-memory multiprocessing systems being developed. Such systems require semaphores to ensure that the multiple processors of the system modify system data structures in a consistent manner.

The i860 supports a general Lock and Unlock instruction pair, which causes the processor to run all of the instructions between them in an atomic manner with interrupts blocked. (Note that the hardware enforces a limit of 32 such instructions.)

The 88000 supports the XMEM instruction, which loads a memory location, tests it for 0, and if a 0 is detected, stores the specified register contents into the memory location. The load/stores are indivisible on the bus.

The Sparc architecture supports two types of semaphore instructions (though early implementations only support one). The Load-Store Unsigned Byte instruction reads a memory location and then writes that memory location to all 1s in an atomic manner. The Swap instruction causes a memory location to be read and then replaced with the contents of a specified register.

In comparison, it would appear that the i860 Lock/Unlock mechanism provides better support for such things as counting semaphores. However, in fact, the actual number of instructions required to implement such a construct (and therefore the speed to execute it) is approximately the same for all three architectures. Both general mechanisms, the Sparc/88000 and the i860, require multiple instructions to obtain a lock, increment or decrement the semaphore, and then re-

lease the lock. None of the three architectures provides a single-instruction implementation as in the IBM S/370.[5]

Two potential, but small, benefits of the i860 mechanism in an application using the Unix operating system exist. One is its ability to spin on the lock at a low SPL level (interrupt level), and the other is its ability to perform short semaphore or other operations without raising the SPL level at all. In the first case, the Unix kernel requires that the SPL level be raised before attempting to obtain a lock that could also be required at a higher interrupt level. This requirement normally means that software on a processor such as the 88000 or Sparc must raise the SPL level to ensure that it does not get interrupted after obtaining the lock. (If it were interrupted, a deadlock situation could arise.) However, since the i860 Lock/Unlock mechanism blocks interrupts, the SPL level does not have to be raised until the lock has been successfully obtained. In addition, if the work performed on the semaphore or the desired code is short enough (less than 32 instructions), the i860 mechanism allows the software to keep the SPL level the same. In total, however, both of these benefits are small and not of sufficient size to consider further.

*Multiply/divide.* Of the three architectures, only the 88000 provides both of the basic integer multiply and divide instructions. The i860 architecture supplies a multiply operation via its FMLOW floating-point operation but provides a library routine for division. The Sparc architecture, alternatively, provides a Multiply Step instruction and library routines to implement both multiply and divide operations. The lack of these instructions constrains the i860 and Sparc architectures in measurably increasing multiplication and division performance by using any hardware available in future implementations. As such, i860 and Sparc implementation performance will suffer on applications that require extensive multiplication and division operations unless vendors add the basic multiply and divide instructions to the architecture.

**Branches.** The three architectures have the concept of a delayed branch. Here the instruction sequentially following the branch executes independently of whether the branch is or is not taken. This feature increases performance of pipeline implementations by reducing the flushing effect of branches on the pipeline. Studies have indicated that this technique is successful in eliminating the branch penalty in 60-70 percent of the cases.[6]

In addition, the three architectures have the ability to essentially annul the execution of the instruction in the delay slot. This provision eliminates the potential increase in code size identified after having to fill the delay slot with a NO-OP instruction. Avoiding this increase reduces the factor by which the RISC code size will increase over a traditional CISC architecture.[7]

None of the three architectures incorporates branch prediction in the instruction set as in the AT&T Crisp[8] architecture. Such software prediction would reduce the branch penalty. However, all of the architectures could adopt any one of the many hardware branch-prediction strategies for a particular implementation.[9] While studies have shown that software branch prediction may be more cost effective to implement, the hardware schemes are not excessively expensive and do provide very good branch prediction.[9]

*Additional comparison and looping support.* In addition to the usual branch instructions, the i860 architecture provides additional support for those loop operations that terminate with a comparison against 0 via the BLA (branch on loop condition code and add) instruction. This single instruction decrements a counter, compares it to 0, and then branches on that comparison—all in one cycle.

In comparison, the 88000 and Sparc architectures require two instructions (and two cycles) to implement the same functionality. In the 88000 architecture the first instruction decrements the counter. Meanwhile the second instruction compares the result against 0 (creating an intermediate set of condition codes) and executes the branch operation. In the Sparc architecture the first instruction decrements the counter (and sets the condition codes). The second instruction executes the branch operation (based on the condition codes).

However for loops not terminated by a test against 0, all three architectures require a total of three instructions to perform the decrement, comparison, and branch operations. Studies have shown that while loops with a termination of 0 are common, they are not the predominant case.[10] Therefore though the i860 provides better performance in this case, the total performance improvement overall will not be large.

*Condition codes.* The three architectures support condition codes on which some or all of their branch instructions perform a test.

The i860 provides both the traditional condition-code approach and the loop control instruction just described, which uses a separate condition code. Unlike the Sparc and 88000, however, the i860 arithmetic instructions always set the condition codes. This specification makes the implementation of more complicated pipeline schemes supporting out-of-order execution and multiple-instruction executions per cycle more difficult to implement.

The 88000 architecture departs from the traditional approach of condition codes held in the processor status word. Instead it writes status information resulting from a Compare operation in a general register specified in the Compare instruction. Conditional branch instructions correspondingly test the specified general register to determine whether the branch operation should proceed. Given that no separate condition codes

**Five different addressing modes can be synthesized by each architecture.**

exist, future implementations of the architecture will more easily employ complicated pipelining schemes supporting out-of-order execution and multiple instruction execution per cycle.

The Sparc architecture allows many instructions to set the condition codes. In addition it provides an explicit Compare instruction and all of its branch instructions test the condition codes. Arithmetic instructions can optionally set the condition codes or leave them unaffected. These provisions will enable future implementations of the architecture to more easily employ the same pipelining schemes as described for the 88000.

While the traditional method has offered separate condition codes, arguments have been put forth against condition codes. They add difficulties to the hardware design and result in an unorthogonal instruction set. The 88000 addressed certain concerns[11] by having the condition-code bits stored in any specified register, as described earlier. This requirement minimized any hardware implementation problems and facilitated the hardware support of parallel integer and floating-point operations. It also effectively eliminated yet another of the few registers that are available to the user. However, given the magnitude of the difficulties associated with using condition codes, any additional hardware that may be required would be small.

**Addressing modes.** The three architectures share two basic addressing modes for operand access. They are base + offset and base + index. With register 0 returning 0 all the time, five different addressing modes can actually be synthesized. They are:

- *register:* R*x*, where *x* is the register number;
- *register indirect:* (R*x*), where *x* is the register number;
- *register indirect with index:* (R*x*, R*y*), where *x* and *y* are the register numbers;
- *register indirect with immediate offset:* offset(R*x*), where *x* is the register number; and
- *immediate, signed and unsigned.*

In the register indirect with immediate offset mode, the i860, 88000, and Sparc support 16-bit signed offset, 16-bit unsigned offset, and 13-bit signed offset forms, respectively. Given that long immediate offset mode is rarely used, the difference in the length of immediate offset mode is irrelevant. However, support of the signed immediate mode provides some extra flexibility over the unsigned immediate mode.

In the immediate mode, the i860 architecture supports the 16-bit signed immediate form for arithmetic operation and 16-bit unsigned immediate form for logical operation. The 88000 architecture supports the 16-bit unsigned immediate form. The Sparc architecture supports only the 13-bit signed immediate form. Given that long immediate modes are rarely used, the difference in the length of immediate modes is irrelevant. However, the support of the signed immediate mode provides some extra flexibility over the unsigned immediate mode.

It is interesting to note that the above addressing modes are also the five most frequently used addressing modes in CISC machines.[12, 13] In fact, the least frequently used address mode of the five, register indirect with index, has a frequency of only 6 percent.[13]

The 88000 also supports index mode with scaling. This addressing mode simplifies index computation for accessing halfword arrays as well as word arrays. The addressing mode is useful for artificial intelligence languages and scientific computing. However, it will have a low frequency of usage in a general-purpose computing environment. Hence, little performance gain will be seen.

To eliminate the requirement of an additional read port to its register file, the i860 memory store instruction does not support the use of register indirect with index mode. This absence of support introduces asymmetry to the instruction set and hence an exception to the compiler. However, based on a CISC-machine study,[14] less that 4 percent of the second operand and the destination operand in a triadic operation use the address mode. Therefore, we see very little performance impact for the lack of it. For floating-point vector-processing performance, the i860 supports the autoincrement mode for constant stride vector addressing. Since very little floating-point vector processing occurs in general-purpose computing, we again see very little performance impact.

*Control-transfer address.* All three architectures provide two addressing methods for control-transfer operations, PC-relative and register indirect. For PC-relative conditional transfer, the i860 provides 16-bit and 26-bit offset modes, the 88000 provides 16-bit offset, and the Sparc provides 22-bit offset. The i860 offers a better range for PC-relative transfers. However, based on the previously mentioned CISC-machine study, a 16-bit offset mode sufficiently processes 93 percent of PC-relative branches. A 15-bit offset mode is sufficient for 87 percent of PC-relative branches.[14]

Given the code expansion due to the RISC architecture and the trend in program-size growth, a 16-bit offset mode will probably be good for close to 87

**All three architectures provide more registers than their CISC forebears.**

percent of all PC-relative branches. Since 15-20 percent of the instructions executed are nonprocedure call-related, PC-relative control-transfers, only 2 percent additional branches are needed to reach the branch target. The penalty of a shorter 16-bit offset mode is insignificant.

For unconditional transfer and procedure call or return, the three architectures provide both register indirect and PC-relative addressing modes.

**Registers.** The number of application-usable registers becomes a key factor in the performance of RISC processors, given the relative performance penalties associated with accessing variables in cache and/or main memory. This factor and the increasingly sophisticated register-allocation schemes of today's compilers form the primary driving forces behind incorporating a larger set of registers into the architectures of current processors.[15, 16] In this area, all three of the architectures provide substantially more registers than their CISC forebears. However, the registers differ in the way they are used and the number that are available.

The 88000 is the weakest in this area with only thirty-two 32-bit registers for both integer and floating-point operations. Given that each floating-point operand typically takes two registers, the effective number of values that can be contained in the register file is much less than 32. In comparison, the i860 and Sparc with thirty-two 32-bit integer registers and an additional thirty-two 32-bit floating-point registers can hold a substantially larger number of values in the register file. Studies have indicated that this increased number of registers should result in better performance for the i860 and the Sparc.[15]

In addition to the 32 integer registers directly addressable via the instruction set, the Sparc architecture also supports a register-windowing system. This system provides between two and 32 windows of registers arranged as a circular buffer. (For a detailed explanation see the *Sparc Architecture Manual* and Patterson and Sequin.[17, 18])

Proponents of this and similar register-windowing schemes argue that the windowing provides a number of benefits. Among them are:

1) The compiler does not have to save/restore registers across function calls, thereby increasing the speed of the function calls.

2) The compiler does not have to be as complex because it does not have to perform sophisticated register allocation.

3) The windowing system provides a mechanism for providing an increased number of windows in a user software-transparent manner.

Meanwhile, detractors argue that windowing has potential drawbacks:

1) The overflow or underflow of the circular buffer (running out of usable windows) requires that some portion of the windows must be flushed or filled.

2) Context switches now involve the save/restore function of significantly more registers than in the traditional case.

The exact value of a register-windowing scheme (such as that supported by Sparc) in comparison with the use of sophisticated register-allocation techniques (such as those used by the i860 and 88000) has been the subject of several investigations.[16, 17, 19] The studies show that the relative performance of the two options is essentially equal and that the register-windowing scheme provides better performance in some cases.

The relative disadvantages of the register-windowing scheme turn out to be few because the frequency of overflows/underflows and context switches is small in comparison with the frequency of procedure calls. However, not all cases achieved the relative advantages of the register-windowing scheme due to the newer, sophisticated approaches to register allocation that take advantage of program characteristics (such as the high percentage of time spent in leaf procedures).

In addition to these architectural aspects, a major contention of the proponents of register allocation is that the implementation of a register window-based architecture will suffer from having to support the register windows. In particular, they point out that the frequencies of two implementations (one having register windows and one having a typical register file) will not be the same given equal technology because the register windows will require additional logic in the critical path. While this contention has yet to be proven (current Sparc implementations run at frequencies just as fast or faster than the i860 and 88000 frequencies), it could affect certain implementations. However, architectures with register-windowing schemes can support any number of windows including one (same as the register allocation approach) or two (depending on the exact implementation, for example, Sparc requires two). Any negative effects of windowing in such an implementation could be reduced or eliminated as necessary by reducing the window count to a low level. (Though any "old" code, presumably compiled without sophisticated register allocation, would run poorly in such an implementation.)

**Byte ordering.** The i860 and the 88000 support byte-ordering formats called big endian and little endian. The Sparc supports the big-endian format. The selec-

tion of the byte-ordering method becomes a data-compatibility issue with existing architectures. The Sparc architecture originated at Sun Microsystems Inc., where most of the products are Motorola 680X0-based (big-endian byte order). Big-endian format thus becomes the logical choice. Similarly, the Intel 80X86 line supports the little-endian format, a logical choice therefore for the i860.

The i860 and 88000 support both byte orderings statically. As a result, data can be exchanged with a big-endian machine or a little-endian machine without reversing the bytes or changing the byte numbering. Thus, the i860 and 88000 provide a migration path for data and databases generated from machines of either byte orderings. However, the 88000 ABI specifies the big-endian format (Motorola's 680X0 format) for interfacing to the operating system. Any application running in little-endian byte order must somehow swap the bytes to interface to the operating system. It is not yet clear what byte order the i860 ABI will specify. However, to maintain some sort of data compatibility with the Intel 80486 line, the i860 ABI will probably adopt the little-endian format. Again, any application running in big-endian byte order must somehow swap the bytes to interface to the operating system.

Note also that none of the three architectures provides a complete data-compatibility solution. The majority of the existing machines supports arbitrary byte alignment for data, whereas all three architectures do not. Considering the cost of breaking instruction compatibility (migrating from CISC to RISC), the data incompatibility issue is minor.

**Data types.** The three architectures supply the usual set of integer data types, namely, byte, unsigned byte, halfword, unsigned halfword, word, and unsigned word.

The three architectures also supply the usual set of ANSI/IEEE floating-point data types, namely, single-precision and double-precision.[20] In addition, the Sparc supports extended-precision floating-point operations, giving it an edge for applications requiring additional precision. While current language standards do not support extended-precision floating-point data, note that as RISC implementations approach mainframe performance the demand for extended-precision floating-point data will increase.

For different target markets, the three architectures support additional data types. The i860 supports 8-bit, 16-bit, and 32-bit pixels to provide high-performance 3D graphics processing. The 88000 supports bit-field data. However, it is limited to data within a word. It has a much narrower range of applications than the Motorola 68020 bit-field instructions that operate across word boundaries. The Sparc supports tagged data. The support of this data type has been shown to provide a 10-25 percent execution-time savings for systems using dynamic data typing, for example, Smalltalk.[21] Since these special data types are really targeted for specific applications, the support of such data types and related operations will not have any performance impact on general-purpose computing.

**Floating-point arithmetic.** The three architectures support the *ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*[20] through different levels and mixes of hardware and software emulations. They supply the usual set of floating-point instructions, namely, load/store, integer to floating point, floating point to integer, add, subtract, multiply, and compare.

The Sparc and the 88000 supply division and square-root instructions, whereas the i860 supports the division and square-root functions via reciprocals, a similar approach taken by Cray supercomputers. Here, a Newton-Raphson iterative sequence using the multiply and reciprocal instructions performs a division or square-root operation. As a result, i860 implementations will suffer on those applications that require extensive division and square-root operations. However, in general, these operations have low usage frequencies. Measurements taken from an execution of the SPICE circuit simulator on an MOS memory cell circuit show that floating-point arithmetic occurs only 12 percent of the overall time.[22] Out of that 12 percent, division occurs only 9 percent of the time. In other words, the overall usage is 1 percent.

The i860 floating-point architecture supports both scalar and pipelined modes. However, the pipelines are exposed. This means that either software compatibility may have to be broken in the future or a restriction be placed on future implementations.

The i860 also has a set of instructions that can initiate an add/subtract and a multiplication, and control the data paths between the adder and the multiplier pipelines. Vector operations, like multiply and accumulate, can be synthesized (by controlling the data paths accordingly) and be speeded up considerably. However, it is questionable how well a compiler can vectorize and make use of the exposed pipeline. To take full advantage of the vector processing, an application programmer will probably have to make calls to a hand-coded library of vector-processing routines. Again, the i860 vector/pipeline operations are useful for a particular market, and we see little vectorizing/performance for general-purpose use.

**Memory management.** The three architectures support fairly traditional memory management architectures though each provides additional support in many crucial areas. All three architectures support a full 4-Gbyte virtual address space.[23-25] While this space will be sufficient in the near term, all three will have to deal with larger virtual address spaces in the longer term (a la HP Spectrum[26] and IBM 801 and PC RT[27]). In all three cases, retaining compatibility will be a major architectural challenge.

The i860 supplies a 4-Gbyte physical address spectrum, while the 88000 supports an 8-Gbyte spectrum and the Sparc supports a 64-Gbyte spectrum. The i860 and Sparc share their respective address spaces between the user and the operating system with the exact boundary not being fixed in hardware. Alternatively, the 88000 hardware divides the 8-Gbyte space into 4 Gbytes reserved for the operating system and 4 Gbytes for the user. The ability to directly address spaces of greater than 4 Gbytes will become increasingly important in future systems with multi-gigabyte main memories and with 32-bit, direct-addressed input/output buses. Sparc sufficiently addresses this need with its 64-Gbyte address space, while both the 88000 and i860 restrict space to more traditionally sized physical address spectrums. Fortunately, both the i860 and 88000 have reserved bits in their page table entries. These bits could be used to increase their physical address spectrum in the future.

The 88000 and the i860 support two levels of address translation while the Sparc supports three-level translation. Theoretically, the two-level translation will reduce the time to translate the virtual address into a physical address when the translation cache or the lookaside buffer does not contain the translation information. However, the overall effect is small due to the high TLB hit rates. A detrimental effect of only having two levels of translation, on the other hand, is the overhead (in terms of the number of pages required) encountered to map the large, sparse address spaces of processes in the Unix operating system. The adoption of Unix System V Release 4.0 along with the increased number of logical segments used in applications (shared libraries, mapped files, etc.) makes it increasingly important to reduce the overhead of the page tables associated with each process.

All of the architectures support a 4-Kbyte page size. While larger than many page sizes in traditional CISC architectures, the increased size of applications (as well as the increased size of RISC-executable files) justifies the use of a large page size. Even larger page sizes (more than 4 Kbytes) are good for systems with a large amount of memory and running relatively few large applications (workstations). They are not suitable for systems with a small amount of memory and running numerous small applications. For a system with a fixed amount of memory, for instance 8 Mbytes, a 4-Kbyte page size results in a "pool" of 2,000 pages. An 8-Kbyte page size results in a pool of only 1,000 pages. For applications with a large number of small processes, higher performance will be achieved with systems holding 2,000 pages in the pool rather than 1,000.

Given the small number of TLB entries available in the microprocessor implementations of these architectures, only a small amount of virtual address space can be mapped without incurring TLB miss penalties. If only pages are supported in a memory management architecture, a typical TLB implementation with 64 entries will map only 64 × 4 Kbytes, or 256 Kbytes of memory. While such a mapping size is sufficient for most user applications with their high degree of locality, it is not enough for large applications or the Unix kernel, which exhibit a very low degree of locality. Therefore, support of some larger form of mapping, for example, segments, is required to provide sufficient performance. In addition, such large mappings require large, continuous pieces of physical memory. Many applications such as the Unix kernel really use only a portion of multiple mappings (for text and stack). Therefore, it is important that the mappings not be too large to minimize the wastage of physical memory. (Though some of it can be effectively used by double-mapping this area of physical memory.)

Both the Sparc and 88000 architectures support such a larger mapping. The 88000 supports 4-Mbyte mappings with the option to individually enable or disable 256-Kbyte "chunks" of that mapping. The Sparc, alternatively, supports 256-Kbyte, 16-Mbyte, and 4-Gbyte mappings. The ability of both architectures to effectively map 256-Kbyte pieces of the address space sufficiently addresses the problem of the low locality and at the same time minimizes the wastage of physical memory.

The i860, however, does not support any form of larger mappings. This deficiency will result in a much lower effective TLB hit rate, which could severely impact overall system performance in some applications. Support of some kind of large mapping facility could be added, however, since this feature is typically not visible to the user and is hidden by the kernel (the virtual memory subsystem in Unix V Release 4.0). Also, the most crucial application of the larger mapping appears for the kernel when a change from pages to a larger mapping would be entirely invisible to the user.

The three architectures provide the minimum user/kernel and read/write protections. Sparc, in addition to these minimum permissions, also offers a limited combination of Execute permissions. The addition of Execute permissions provides Sparc with capabilities that will be useful in dealing with such things as dynamic shared libraries.

Overall, the i860, 88000, and Sparc memory management architectures provide essentially equal capabilities with the exception of the lack of large mapping support in the i860. The Sparc architecture offers the most flexibility and possibilities for future growth. But all three architectures will require significant upgrades when virtual address spaces of greater than 4 Gbytes become important.

In summary, examination of the various components of the overall architectures reveals that each have some areas that offer better support than the others and some areas that provide worse support. Table 2 summarizes the assessments of the various components

**Table 2.**
**Relative Architecture Support.**

| Area | i860 | 88000 | Sparc |
|---|---|---|---|
| General | | | |
| Unique instructions | ≥ | ≥ | ≥ |
| Semaphores | = | = | = |
| Multiply/divide | ≤ | = | ≤ |
| Branches | ≥ | = | = |
| Addressing modes | = | = | = |
| Registers | = | ≤ | = |
| Data types | = | = | = |
| Floating-point functions | ≤ | = | = |
| Memory management | ≤ | = | ≥ |

of the architectures that we examined, the i860, the 88000, and the Sparc. For each of the components in the table, we indicate whether we found that the architecture was slightly inferior with respect to the others (≤), essentially equal to the others (=), or slightly superior to the others(≥).

The i860 architecture is weaker in the floating-point area because of the software-visible pipelines, in the memory management area because of its lack of support of a large memory mapping, and in the higher math area due to its lack of a full divide instruction. However, the i860 architecture is stronger in the branch area because of its loop control support instruction. The 88000 architecture is weaker in the area of registers because of the smaller number of registers that the architecture supports. The Sparc architecture is weaker in the area of higher math functions due to its lack of support for full multiply and divide instructions. However, the Sparc architecture is stronger in the memory management area because of its more flexible MMU, or memory management unit, and additional page permissions.

Of the relative weaknesses that were identified, they vary in how difficult they would be to change. The lack of a large mapping in the i860 could be remedied by the addition of such a construct to the MMU. Since this construct will most importantly be used by the kernel, its addition could be made entirely user-software transparent. The software visibility of the floating-point pipelines in the i860, alternatively, most likely cannot be addressed without significantly breaking software compatibility. As in the Sparc case, the addition of a full divide instruction could be added fairly easily.

The number of registers supported in the 88000 architecture would be very difficult, if not impossible, to increase because of the lack of extra, unallocated, bits within the instruction encodings. The lack of full multiply and divide instructions in the Sparc architecture could be fairly easily addressed using an available free opcode number. Such a change could provide both forward and backward software compatibility (assuming the old implementations trapped onto the new instruction). However, new code would run at unacceptably slow rates on old implementations.

In addition to their general support of typical architectural features, each architecture will provide particular applications with much better support than the others due to special architectural features.

1) The i860 provides the best graphics support with its pixel instructions and data types.
2) The 88000 offers the best bit-manipulation support.
3) The Sparc provides the best artificial intelligence support with its tagged arithmetic instructions.

From a system implementation point of view, the three architectures support the basic primitives necessary to implement a general-purpose Unix system implementation. While the primitives may be somewhat more primitive that those in traditional CISC architectures, they do provide the basic building blocks upon which a Unix system can be based. In fact, since the building blocks are relatively primitive, they avoid locking in a particular implementation. For example, a CISC context switch instruction gives an implementation the freedom necessary to create a more optimal solution.

In considering all the factors, we find that no one of the three architectures is clearly inferior or clearly superior to the other architectures. A particularly bad or a particularly good implementation of any of these three architectures will more than make up for any architectural differences that have been identified.

## References


1. G.A. Blaauw, *Digital System Implementation*, Prentice Hall, Englewood Cliffs, N.J., 1976, p. 2.
2. H.S. Stone, *High Performance Computer Architecture*, Addison-Wesley, Reading, Mass., 1987, pp. 1-20.
3. J.D. Foley and A. van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, 1982.
4. W. Stalling, "Reduced Instruction Set Computer Architecture," *Proc. IEEE*, Vol. 76, No. 1, CS Press, Los Alamitos, Calif., Jan. 1988, pp. 38-55.
5. *IBM System/370 Principles of Operation*, 9th ed., International Business Machines Corp., Poughkeepsie, N. Y., Oct. 1981.
6. D.J. Lilja, "Reducing the Branch Penalty in Pipelined Processors," *Computer*, Vol. 21, No. 7, July 1988, pp. 47-55.
7. J.A. DeRosa and H.M. Levy, "An Evaluation of Branch

Architectures," *Proc. 14th Ann. Symp. Computer Architectures*, CS Press, June 1987, pp. 10-16.

8. D.R. Ditzel and H.R. McLellan, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero," *Proc. 14th Ann. Symp. Computer Architecture*, CS Press, pp. 2-9.
9. J.K.F. Lee and A.J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, Jan. 1984, pp. 6-22.
10. M.G.H. Katevenis, "Reduced Instruction Set Computer Architectures," MIT Press, Cambridge, Mass., 1985.
11. J. Hennessey et al., "Hardware/Software Tradeoffs for Increased Performance," *ACM Proc. Symp. Architectural Support for Programming Languages and Operating Systems*, Mar. 1982, pp. 2-11.
12. C.A. Wiecek, "A Case Study of VAX-11 Instruction Set Usage for Compiler Executions," *ACM Proc. Symp. Architectural Support for Programming Languages and Operating System*, Mar. 1982, pp. 177-184.
13. D.W. Clark and H.M. Levy, "Measurement and Analysis of Instruction Use in the VAX-11/780," *ACM/IEEE Proc. 9th Ann. Symp. Computer Architecture*, 1982, pp. 9-17.
14. B.L. Peuto and L.J. Shustek, "An Instruction Timing Model of CPU Performance," *Proc. Fourth Ann. Symp. on Computer Architecture*, Mar. 1977, pp. 165-178.
15. F. Chow and J. Hennessey, "Register Allocation by Priority-based Coloring," *Proc. ACM SIGPlan 84 Symp. Compiler Construction*, June 1984, pp. 222-232.
16. D.W. Wall, "Register Windows vs. Register Allocation," *Proc. ACM SIGPlan 88 Symp. Programming Language Design and Implementation*, June 1988, pp. 67-78.
17. *Sparc Architecture Manual*, Sun Microsystems, Inc., Mountain View, Calif., 1987.
18. D.A. Patterson and C.H. Sequin, "A VLSI RISC," *Computer*, Vol. 15, No. 9, Sept. 1982, pp. 8-21.
19. R.P. Colwell et al., "Computers, Complexity and Controversy," *Computer*, Vol. 18, No. 9, Sept. 1985, pp. 8-19.
20. *ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*, CS Press, 1985.
21. D.M. Ungar, *The Design and Evaluation of a High Performance Smalltalk System*, MIT Press, Cambridge, Mass., 1986.
22. W. Hollingsworth, H. Sachs, and A.J. Smith, "The Clipper Processor: Instruction Set Architecture and Implementation," *Comm. ACM*, Feb. 1989, pp. 200-219.
23. *The Sparc Reference MMU Architecture, Rev. 1.3*, Sun Microsystems, Inc., Oct. 1988.
24. *i860 Programmer's Reference Manual*, Intel Corp., Santa Clara, Calif., Feb. 1988.
25. *M88000 Architecture Specification*, Motorola Corp., Schaumburg, Ill., 1986.
26. M.J. Mahon et al., "Hewlett-Packard Precision Architecture: The Processor," *Hewlett-Packard J.*, Aug. 1986, pp. 4-22.
27. "The 801 Minicomputer," *IBM J. Research and Development*, Vol. 27, May 1983, pp. 237-246.

**Richard S. Piepho**

**William S. Wu**

**Richard S. Piepho** is a member of the technical staff at AT&T Bell Laboratories in Naperville, Illinois. He currently works on the development of future AT&T computer systems. His past work has included performance and architectural analysis of the company's RISC processor, Crisp, as well as the University of California at Berkeley's RISC I processor.

Piepho received a BSEE from Purdue University and an MS in computer science and electrical engineering from the University of California at Berkeley. He is a member of the IEEE Computer Society, Tau Beta Pi, and Eta Kappa Nu.

**William S. Wu** was the AT&T Bell Laboratories architect for the 32-bit WE32200 microprocessor as well as a member of the design team. His interests include very large scale integration architecture, interconnection networks, and multiprocessor architecture. Wu received a BSEE from the University of Minnesota, an MSEE from Carnegie Mellon University, and a PhD from the University of Michigan. He is a member of the IEEE Computer Society, the ACM, Eta Kappa Nu, and Tau Beta Pi.

Questions concerning this article can be addressed to Richard S. Piepho, AT&T Computer Systems, 1100 E. Warrenville Road, Naperville, IL 60566, or William S. Wu, AT&T Data Systems Group, Crawfords Corner, Holmdel, NJ 07733.

## Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

**Low** 156 **Medium** 157 **High** 158