COMPUTER

# Design Decisions in SPUR

**Ten superminicomputer processors for your desk will come packaged as a VLSI workstation called SPUR, once the team at UC Berkeley finds a partner to transfer their upcoming prototype to industry.**

SPUR (Symbolic Processing Using RISCs) is a multiprocessor workstation being developed at the University of California at Berkeley to conduct parallel processing research. Its development is part of a multi-year effort to study hardware and software issues in multiprocessing, in general, and parallel processing in Lisp, in particular.* This article concentrates on the initial architectural research and development of SPUR.

Two key observations motivated the architecture of SPUR. First, although parallel processing hardware has existed for many years, these systems have been difficult to program. Often the architectural features of a parallel machine, particularly the interconnection network between the processors, had to be considered during programming.[1] The complexity of managing such details has left parallel processing a novelty, rather than the norm. Consequently, we are designing SPUR to simplify parallel processing software by providing a single global memory that can be shared, with uniform access times. Implementing a high-performance shared memory system increases the system's hardware complexity, but we believe the shared memory software model facilitates the rapid development of parallel processing software and permits implementation of other, more restricted, sharing paradigms (such as message passing).

Second, hardware is more difficult to design, construct, debug, and modify than most software. Consequently, most SPUR hardware features are simple, frequently-used primitives. We migrate features from software into hardware only if doing so achieves a significant performance gain in return for reasonable design and implementation costs. The complex hardware features included in SPUR either facilitate parallel processing (for example, hardware-based cache consistency) or make large contributions to performance (for example, the instruction buffer and Lisp data-type tags).

The SPUR processor extends the work of reduced instruction set computers (RISC)[2] and Smalltalk on a RISC (SOAR)[3] with some special support for two emerging standards: Common Lisp[4] and IEEE Standard 754-1985 for binary floating-point arithmetic.[5] We designed the Lisp and floating-point support so that software that does not use these extensions is not penalized by their existence. Thus, the SPUR processors are general-purpose processors with some support for Lisp and floating-point, rather than special-purpose Lisp or floating-point processors.

*We distinguish between *multiprocessing* and *parallel processing*. Multiprocessing occurs whenever two or more processors in a computer are used at the same time. Parallel processing occurs when they are cooperating on the same job. All parallel processing is multiprocessing, but not vice versa.



Nov 86

0018-9162/86/1100-0008$01.00 © 1986 IEEE

The SPUR project consists of SPUR workstation development and research efforts in integrated circuits, computer architecture, operating systems, and programming languages. Integrated circuit researchers are examining complementary metal oxide semiconductor (CMOS) design styles, the effects of scaling very large scale integration (VLSI) circuits, and control and clocking issues. Computer architecture researchers are studying multiprocessor address trace analysis, cache consistency, virtually-tagged caches, in-cache address translation, multi-level cache design, coprocessor interfaces, instruction delivery, Lisp hardware support, and floating-point implementations. Operating systems researchers are investigating network file systems, network page servers, the effects of large physical memories on virtual memory implementations, and workload distribution. Programming languages researchers are examining parallel garbage collection algorithms, techniques for specifying parallel programs, and methods of compiling parallel Lisp programs.

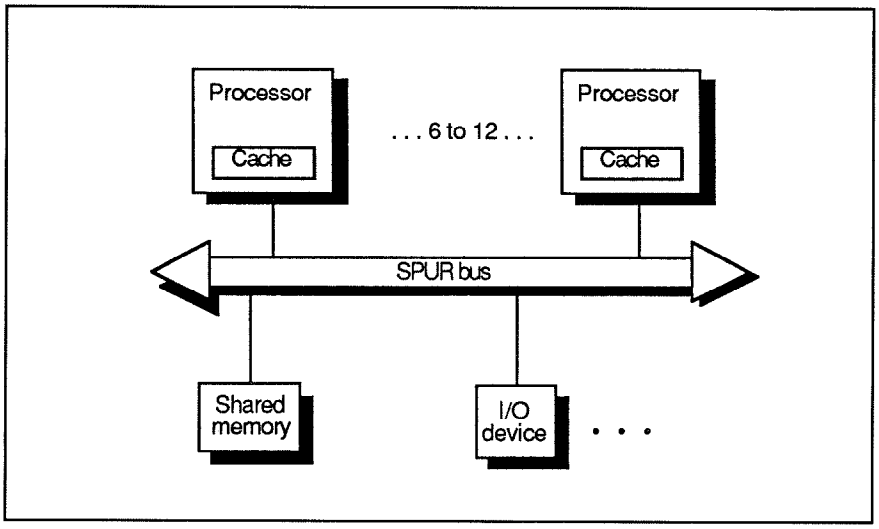


**Figure 1. SPUR workstation system. SPUR is a shared-bus multiprocessor. The system supports several identical high-performance processors on a modified Texas Instruments NuBus. Each of the custom processors contains a large cache to reduce the bandwidth required from the bus and shared memory.**

## System overview

SPUR contains 6 to 12 high-performance homogeneous processors (see Figure 1). The number of processors is large enough to permit parallel processing experiments, but small enough to allow packaging as a personal workstation.

The processors are connected to each other, to standard memory, and to input/output devices with a modified NuBus. Using a commercial bus reduces prototype design time by allowing the use of standard subsystems and memory.

SPUR supports sharing between cooperating processes with a global, shared memory. System performance is improved by placing 128K-byte caches on each processor to reduce bus traffic, memory contention, and effective memory access time. Each of these caches is accessed with virtual addresses, rather than physical addresses, so that address translation is not necessary on cache hits. On cache misses, virtual addresses are translated into physical addresses before accessing shared memory.

The caches are supplemented with hardware that guarantees that copies of the same memory location in different caches always contain the same data. This enables programmers to write soft-


## The Authors:

Mark Hill, Susan Eggers, Jim Larus, George Taylor, Glenn Adams, B. K. Bose, Garth Gibson, Paul Hansen, Jon Keller, Shing Kong, Corinna Lee, Daebum Lee, Joan Pendleton, Scott Ritchie, David Wood, Ben Zorn, Paul Hilfinger, Dave Hodges, Randy Katz, John Ousterhout, and Dave Patterson

The authors are faculty, graduate students, and former graduate students associated with the Dept. of Electrical Engineering and Computer Sciences at the University of California at Berkeley.

The following authors are working on a PhD in computer architecture under the guidance of **Randy Katz** and **Dave Patterson**: **Susan Eggers** (cache coherency), **Garth Gibson** (I/O and bus architecture), **Paul Hansen** (coprocessor architectures), **Mark Hill** (cache simulation methods and analysis; co-supervised by Alan Smith), **Corinna Lee** (floating-point arithmetic), **George Taylor** (Lisp instruction set design), and **David Wood** (memory management).

Authors working with **Dave Hodges** and **Dave Patterson** toward a PhD in integrated circuits are **B. K. Bose** (floating-point arithmetic), **Shing Kong** (CPU microarchitecture), and **Daebum Lee** (CMOS CPU circuit design).

**Jim Larus** and **Ben Zorn** are working with **Paul Hilfinger** in software for PhD dissertations on restructuring Lisp programs for concurrent execution and multiprocessor storage reclamation, respectively.

**Glenn Adams** is working on his PhD in computer-aided design for VLSI.

**John Ousterhout** supervises a group of students in building Sprite, the network operating system for SPUR.

Students who have graduated and moved to industry are **Jon Keller** (Pad design), who received his MS and returned to Hewlett Packard; **Scott Ritchie**, who joined Sun Microsystems after receiving an MS (address translation); and **Joan Pendleton**, who also joined Sun Microsystems after receiving a PhD (VLSI CPU design).

Readers may write to the authors at the Computer Science Division, Dept. of Electrical Engineering and Computer Science, UC Berkeley, CA 94720.

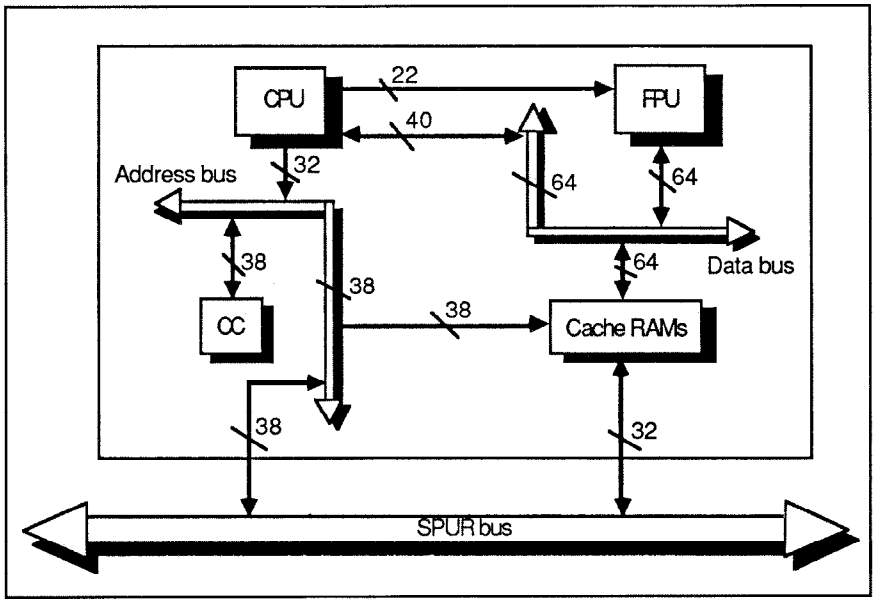


**Figure 2. SPUR processor board. A SPUR processor is implemented on a single board that contains three custom VLSI chips and 200 standard chips. The three custom chips are the cache controller (CC), the CPU, and the floating-point coprocessor (FPU). Standard chips are used to hold the state, address tags, and data of the cache (cache RAMs), and to connect functional components together (not shown). Memory addresses and data are handled on separate buses. The address bus is 38 bits wide to accommodate global virtual addresses. The data bus is 64 bits wide to handle floating-point data. The FPU tracks instructions executed by the CPU with a special 22-bit connection. Some infrequently used data paths are not shown here.**

ware without considering the existence of cache memory.

The high level architecture of SPUR (multiple processors communicating through shared memory over a common bus) is comparable to a few commercial multiprocessors, such as the Sequent Balance 8000 and the Elxsi 6400. However, since neither commercial machine was intended to be a low-cost workstation, the differences in their more detailed architectures are fairly significant.

The Sequent is built from 10-MHz National Semiconductor 32032 central processor unit chips communicating over a pipelined, packet-switched bus. The high speed of the bus (relative to that of the CPUs) enables the Sequent to support between 2 and 12 processors with relatively small caches (8K bytes) and a write-through write policy.

The Elxsi 6400 processors and bus interface logic have been implemented with emitter-coupled logic (ECL) gate arrays. The bus is also packet switched, with a 25 ns cycle time, and easily supports up to eight CPUs. Cache consistency in the 16K-byte, two-way set associative caches is maintained by a variety of software techniques.[6]

## Processor overview

A SPUR processor is a general-purpose RISC processor that provides support for Common Lisp and IEEE floating-point. A processor is implemented on a single board with about 200 standard chips and three custom two-micron CMOS chips: the cache controller (CC), the CPU, and the floating-point coprocessor (FPU). (See Figure 2.)

The CC chip manages the cache. This includes handling cache accesses by the CPU, performing address translation, accessing shared memory over the SPUR bus, and maintaining cache consistency.

The CPU chip is a custom VLSI chip based on the Berkeley RISC architecture.[7,2] Like the RISC II implementation,[8] the SPUR CPU uses a simple uniform pipeline, hard-wired control, and a large register file. It attempts to issue a new instruction every cycle. The SPUR CPU differs from RISC II because of the addition of a 512-byte instruction buffer, a fourth execution pipeline stage, a coprocessor interface, and support for Lisp tagged data.

The final custom chip is the floating-point coprocessor, which supports the full IEEE standard 754 for binary floating-point arithmetic without microcode control. The FPU executes common operations under hard-wired control. Infrequent operations cause traps and are handled by software.

Initial results with small Lisp benchmarks show that a single SPUR processor is comparable to the VAX 8600 CPU and the Symbolics 3600 CPU[9] (see Table 1).

## The memory system

The SPUR memory system appears to software as flat, global, shared memory, but is implemented with a hierarchy of levels. The fastest level, the *instruction buffer*, is an instruction cache on the CPU chip. The second level, the *cache*, is a cache on the processor board for instructions and data. If information is not found in either of these local memories, then the virtual address is translated into a physical address and a global memory access is made via the SPUR bus. Both the virtual and physical addresses are transmitted on SPUR bus transactions. Off-the-shelf memory and I/O controllers use the physical address. Other cache controllers use the virtual address to preserve software's view of global, shared memory.

**The memory model.** SPUR presents software with a 256G-byte global virtual address space, divided into 256 1G-byte segments. Every process has direct access to four segments via a 32-bit process-specific virtual address. (The segments of a SPUR process resemble VAX-11 regions.) This address is mapped into a 38-bit global virtual address in parallel with the first part of a cache access (Figure 3).

A process's four segments will normally be used for system code and data, user code, a private stack, and a shared heap. Two or more processes that want to share information must share an entire segment. Support for sharing at the granularity of a segment is a compromise between using a single shared virtual address space and supporting sharing of arbitrary-size objects at this level. We rejected the former extreme because it does

not permit hardware-guaranteed isolation of unrelated jobs. We rejected the latter because it is not clear that the benefits justify the hardware cost.

The memory system, except the instruction buffer, uses global virtual addresses instead of process-specific virtual addresses so that information can be manipulated independently of processor and process identifiers. For this reason, cache flushes are not necessary on a context switch or when a process is migrated to a different processor.

We set the global virtual address space limit of 256 segments by balancing the projected needs of software against the cost of hardware implementation. The segment limit does not constrain the number of *lightweight* processes,[10] which use the address space of their parent. This is important because we expect the parallel processing Lisp system to make extensive use of lightweight processes. This limit does, however, restrict the number of concurrently active heavyweight processes (such as Unix shell processes) depending on the number of segments shared. The limit ranges from 64 processes with no sharing to 253 processes with three segments shared.

**The instruction buffer: an on-chip instruction cache.** The instruction buffer is a 512-byte instruction cache on the CPU chip. It reduces contention for the cache so that data references can use the single cache port without stalling the execution pipeline. By enabling instruction fetches and data references to be satisfied in parallel, the instruction buffer creates the illusion of a second cache port.

The instruction buffer also reduces effective instruction access time. This effect has little importance in SPUR because the cache can be accessed in approximately one cycle. Nevertheless, this effect will become increasingly important as technological improvements reduce cycle times faster than inter-chip communication times, thereby making it difficult to access off-chip caches in a single cycle.

The instruction buffer caches 128 32-bit instructions in 16 direct-mapped blocks. Each block contains eight instructions, divided into eight single-instruction *sub-blocks*[11,12] (see Figure 4). Preliminary estimates show that the instruction buffer satisfies at least 75 percent of instruction fetches without cache accesses.[13]

**Table 1. Gabriel benchmark results.**

This table presents execution times for Gabriel benchmarks on the DEC VAX 8600, Symbolics 3600 with instruction fetch unit, and a single SPUR processor. The times for the 8600 and the 3600 are from Gabriel's *Performance and Evaluation of Lisp Systems*.[10] The preliminary SPUR times are gathered with a functional-level simulator of a single processor, assuming a 150-ns cycle time, single-cycle access to a 128K-byte cache, and 15-cycle cache miss time.

The last two columns compare SPUR with the 8600 and 3600. SPUR is slower for the ratios shown in bold. The geometric mean is used to combine the ratios in a manner that gives each benchmark equal weight. Garbage collection time is not included for any of the machines. The 8600 results were gathered with data-type declarations to reduce run-time type checking.

| | Execution time (seconds) | | | Execution time ratio | |
|---|---|---|---|---|---|
| Gabriel benchmark | DEC 8600 | Symbolics 3600 | SPUR (projected) | 8600/ SPUR | 3600/ SPUR |
| boyer | 12.18 | 9.40 | 4.47 | 2.72 | 2.10 |
| dderiv | 6.58 | 3.89 | 1.13 | 5.82 | 3.44 |
| deriv | 4.27 | 3.79 | 0.99 | 4.31 | 3.83 |
| destru | 2.10 | 2.18 | 0.46 | 4.57 | 4.74 |
| div2 | 1.65 | 1.51 | 2.77 | **0.60** | **0.55** |
| fft(single) | 9.08 | 3.87 | 9.47 | **0.96** | **0.41** |
| frpoly(bignum) | 1.40 | 2.10 | 7.17 | **0.20** | **0.29** |
| frpoly(fixnum) | 4.13 | 2.65 | 1.76 | 2.35 | 1.51 |
| frpoly(flonum) | 5.84 | 3.04 | 2.57 | 2.27 | 1.18 |
| puzzle | 15.53 | 11.04 | 7.47 | 2.08 | 1.48 |
| stak | 1.41 | 2.30 | 1.00 | 1.41 | 2.30 |
| tak | 0.45 | 0.43 | 0.13 | 3.46 | 3.31 |
| takl | 2.03 | 4.95 | 1.01 | 2.01 | 4.90 |
| takr | 0.81 | 0.43 | 0.23 | 3.52 | 1.87 |
| traverse | 46.77 | 41.71 | 18.12 | 2.58 | 2.30 |
| triangle | 99.73 | 116.99 | 66.55 | 1.50 | 1.76 |
| geometric mean | | | | 1.97 | 1.73 |

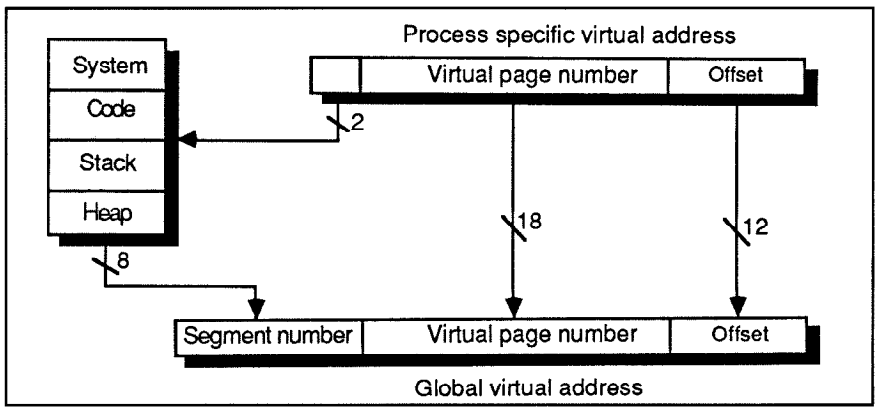


**Figure 3. Virtual memory structure. All processes access virtual memory using a 32-bit process-specific virtual address. This address is converted into a 38-bit global virtual address during the cache lookup. The high-order two bits of the process-specific virtual address are used to select one of four segments from the 256 segments (eight bits) in the global virtual address space. The other 30 bits are used directly for the displacement within the selected 1G-byte segment.**

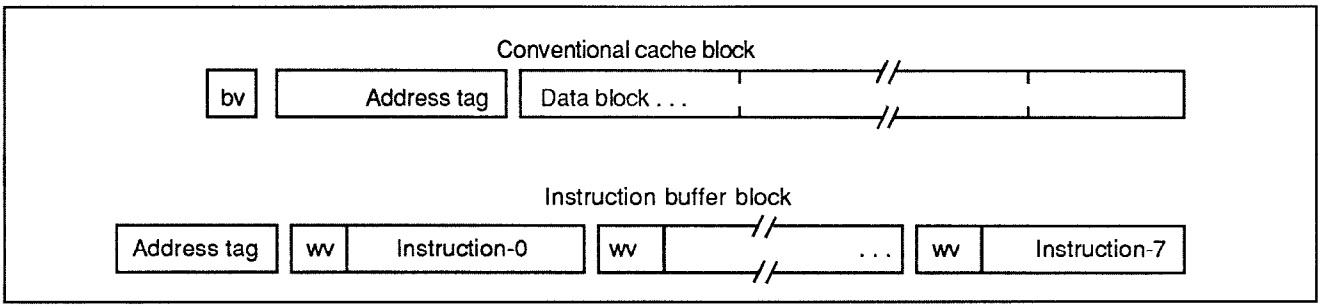


**Figure 4. Cache block with sub-blocks. A conventional cache block (top) has three parts: the cache state bits (labeled *bv* for *block valid* bits), the address tag, and the cache data block. The state bits record whether the information in the block is valid (or dirty for a data cache); the address tag holds the block's memory address; and the data area holds the information cached. The instruction buffer's block (bottom) is divided into eight single-instruction sub-blocks (labeled *instruction-0* to *-7*). Additional state bits, called *word valid* bits (labeled *wv*), are associated with each instruction so that only a single instruction, instead of the entire block, must be loaded on a miss.**

The instruction buffer handles misses differently than most caches. Instead of loading an entire block on each miss, it loads only the fetched instruction. The advantage of fetching single-instruction sub-blocks is that a miss can be completed more quickly. The disadvantages are that the state bits must be extended to include a valid bit for each sub-block in the block and control must handle both block misses (the address tag does not match) and sub-block misses (the tag matches, but the sub-block is not valid). We believe the advantage of using sub-blocks justifies the small increase in chip area and design time needed to implement them.[12]

The instruction buffer miss ratio is reduced by using prefetching from the cache to create the illusion that the entire 32-byte block is loaded on a miss. For example, near the end of a miss to the third instruction in a block, the prefetcher attempts to prefetch instructions 4, 5, 6, 7, 0, 1, and 2. Unless prefetches are blocked by data references, instructions 4 through 7 will be loaded into the instruction buffer before they will be accessed by the execution unit. If the execution unit fetches an instruction that misses in another block, the prefetcher begins prefetching in that block even if the old block is not completely loaded. The prefetcher never hurts performance because it never replaces instructions already in the instruction buffer or interferes with data references or instruction fetch misses.

**The cache.** A 128K-byte cache for instructions and data on every processor board reduces SPUR bus traffic. For a fixed transfer size, bus traffic can be reduced by increasing a cache's size or complexity (degree of associativity).[14] Memory chip technology makes it possible to build larger, unsophisticated caches with fewer packages than smaller, more complex ones. Consequently, the SPUR cache is simple, although larger than the caches in most mainframes. It is direct-mapped, does not permit unaligned accesses, and uses 32-byte blocks to transfer and cache information. It does not prefetch blocks from memory, because prefetching increases bus traffic. Instruction buffer prefetches that miss in the cache are terminated without a memory reference. Simulation results find cache miss ratios under two percent (see the column labeled "Ideal" in Table 2).[13]

The SPUR cache associates virtual address tags, rather than physical address tags, with blocks of data. A virtually-tagged cache is accessed directly, without address translation. In contrast, physically-tagged caches require that address translation be done before or in parallel with the first part of a cache lookup.

Unfortunately, schemes that permit parallel address translation limit the size of the cache, constrain the mapping of virtual pages to physical page frames, or require support for fast reverse mapping (translating physical addresses back to virtual addresses). For a physically-tagged cache access to proceed in parallel with address translation, the cache must be indexed with bits that do not change in address translation. These bits are usually within the page offset, because systems designers are unwilling to constrain the mapping of virtual pages to physical page frames so that some bits of the virtual page number do not change. The bits used to index a cache will be within the page offset only if the cache size divided by its degree of associativity is equal to or smaller than the page size. Consequently, we believe that as cache sizes increase, virtually-tagged caches will become more commonly used.

Another benefit of virtually-tagged caches is that the address translation time does not affect cache hit time since address translation is necessary only for cache misses. Therefore, address translation can be done more slowly in a system with a virtually-tagged cache than in one with a physically-tagged cache where address translation must be less than the cache access time. SPUR exploits this freedom by eliminating the traditional translation buffer.

Most commercial computers use physically-tagged caches rather than virtually-tagged caches. This is because current commercial architectures include three features that make the implementation of virtually-tagged caches difficult. The rest of this section explains how the use of a single, segmented virtual address space and a dual-address bus allows SPUR to avoid the problems commercial implementors have encountered.

*Problems implementing virtually-tagged caches.* The first problem is handling the virtual address space changes associated with most context switches. A virtual address space change means that virtual addresses refer to new locations. A virtually-tagged cache must guarantee that references to the new virtual address space are not accidentally satisfied by data from the old locations.

Address space changes can be handled in a virtually-tagged cache by flushing old data on every context switch or by at-

taching address space identifiers to cached data. The former method reduces performance for large caches. The latter increases the size of cache address tags and can increase cache complexity if synonyms, described in the next paragraph, are allowed. SPUR avoids the problems of virtual address space changes by using a global segmented virtual address space that does not change after a context switch. This results in an increase in tag size comparable to adding address space identifiers, but permits sharing without synonyms.

The second problem for implementors of virtually-tagged caches is that of correctly handling synonyms (aliases). Synonyms are multiple virtual addresses that map to the same physical address. They present a problem when the same physical location is read into a virtually-tagged cache twice with two different virtual addresses, and then one of the copies is modified. To preserve the programmer's model of memory, the virtually-tagged cache must guarantee that a read with the other virtual address gets the new value.

This problem is hard to solve in a single virtually-tagged cache, and even harder to solve in a multiprocessor system with many such caches. SPUR avoids this problem by disallowing synonyms. Instead, two or more processes share information by putting it in a shared segment at the same displacement (the same global virtual address). Software resolves the location of static, shared information at load-time and uses operating system calls that allocate new storage to establish the location of dynamic, shared information.

The third problem with virtually-tagged caches is updating cache data that is being written by an I/O device using a physical address. In the long run, the problem of mapping physical I/O addresses back into virtual addresses can be avoided by having both I/O devices and memory use virtual addresses.

We rejected this approach in SPUR because we wanted to use off-the-shelf, physically-addressed memory boards. Instead, the SPUR bus associates a virtual and a physical address with most bus transactions. The virtual address is used by other cache controllers for maintaining cache consistency. The physical address is used by memory and I/O controllers. The reverse mapping problem is solved by not permitting an I/O buffer to be cached while it is being written by an I/O device. The operating system can guarantee this by putting the buffer on a non-cacheable page or by flushing the buffer from all caches before I/O begins.

**Table 2. In-cache address translation versus translation buffers.**

This table compares SPUR in-cache translation with translation using translation buffers. The metric used, the aggregate miss ratio, is the number of cache misses plus the number of translation misses divided by the number of processor references. Smaller values of this metric predict better performance if the cost of cache and translation misses are comparable (as they are in SPUR). Numbers in parentheses give the magnitude of the aggregate miss ratio relative to the SPUR in-cache aggregate miss ratio. Three comparisons are made. The first two are application programs running on a VAX-11 under Unix 4.2 BSD. Liszt is an address trace of the Franz Lisp compiler compiling a portion of itself. Vaxima is a trace of an algebraic system executing a representative repertoire of commands. The final trace, MVS, is a series of system calls executed by the MVS operating system on an Amdahl 470 machine.

This table assumes that each of the translation mechanisms are invoked only after a reference misses in the SPUR cache, which is 128K bytes large, has 32-byte blocks, and is direct-mapped. The first alternative, Ideal, sets the aggregate miss ratio to the cache miss ratio and assumes translation without cost. The second alternative, SPUR in-cache, uses the cache to hold PTEs for 4K-byte pages. The last three alternatives use translation buffers to do translation. The third uses half of the VAX-11/780 translation buffer (128 entries, 512-byte pages, and two-way set-associative) because the VAX-11 restricts process and system entries to different halves of the buffer. The fourth uses the VAX 8600 translation buffer (512 entries, 512-byte pages, and one-way set-associative) in the same manner. The last alternative uses the IBM 3033 translation lookaside buffer (128 entries, 4K-byte pages, two-way set-associative). In all but one case, shown bold, SPUR in-cache translation performs slightly better than systems that include translation buffer hardware.

Aggregate miss ratio with identical caches
but alternative address translation mechanisms
metric: (cache misses + translation misses) / references

| Trace | SPUR cache plus translation via: | | | | |
|---|---|---|---|---|---|
| | Ideal | SPUR in-cache | VAX-11/780 TB | VAX 8600 TB | IBM 3033 TLB |
| Liszt | 0.00584 | 0.00610<br>(1.000) | 0.00775<br>(1.270) | 0.00784<br>(1.285) | 0.00614<br>(1.006) |
| Vaxima | 0.01844 | 0.01875<br>(1.000) | 0.02432<br>(1.297) | 0.02404<br>(1.282) | 0.02001<br>(1.067) |
| MVS | 0.01677 | 0.01981<br>(1.000) | 0.02208<br>(1.115) | 0.02287<br>(1.154) | **0.01769**<br>**(0.893)** |

The latter does not imply a complete cache flush because the cache supports flushing of individual blocks.

**Address translation without a translation buffer.** The mapping of virtual addresses to physical addresses is usually maintained in a structure called a *page table*.[15] The appropriate page table entry (PTE) is referenced during the address translation process.

Most computers use a special-purpose cache for PTEs, called the *translation buffer*, to reduce address translation time. Translation buffers are important in systems with physically-tagged caches, which require address translation on every reference.

Fast address translation is less important with SPUR's virtually-tagged cache, because address translation is necessary only on cache misses. Consequently, rather than using a translation buffer, the SPUR address translation mechanism always uses cache accesses to reference PTEs logically in shared memory.[16]

The performance of SPUR *in-cache* translation compares to that with fixed-size translation buffers. Moreover, in-cache translation has two advantages. First, it avoids the design and implementation costs of a translation buffer. Second, it keeps PTEs consistent (translation buffer consistency) without special support.

Because traditional translation buffers are merely special-purpose cache memories, multiprocessors that use them suffer from a translation consistency problem (analogous to the data cache consistency problem). Solving this problem requires an increase in either hardware or operating system complexity. The SPUR in-cache translation mechanism avoids this problem by eliminating the translation buffer and storing the PTEs only in the cache, where they are kept consistent by the regular consistency mechanism.

In-cache address translation is invoked when data referenced is not in the cache. (In this discussion, data refers to instructions and data in contrast to address translation information such as PTEs.) The cache controller performs address translation with the following steps. First, a page table base register and the virtual address of the data are used to construct the virtual address of the PTE. Second, the PTE is read from the cache. Third, the physical page address in the PTE and the page offset from the original virtual address are combined to form the physical address of the data. Fourth, a SPUR bus access for the data is made with both the virtual and physical addresses. Last, the data is loaded into the cache and passed on to the CPU.

On rare occasions, the PTE reference will also miss in the cache. Since SPUR places the first level of page tables in pageable virtual memory, a second translation effort is necessary to service the first-level PTE miss.

The second level of page tables is also in virtual memory and thus may be found in the cache. This level, however, is in nonpageable virtual memory at known locations. The physical addresses of second-level PTEs are computed by the cache controller to end the address translation process if the cache access for the second-level PTE misses. SPUR uses the two-level paging mechanism to reduce the physical memory dedicated to PTEs from 256M bytes to 256K bytes.

In-cache address translation works well for the traces shown in Table 2. Translation performance with in-cache translation is comparable to that achieved with translation buffers. In addition, other results show that the presence of PTEs in the cache does not significantly affect cache performance for data (non-PTE) references. The data miss ratio for Vaxima increased by only 0.00004, from 0.01844 to 0.01848. The increase for MVS was larger than with Vaxima, but still not significant (0.00142, from 0.01677 to 0.01819).

**Cache consistency hardware.** The problem of maintaining the shared memory model in multiprocessor systems with cache shared, writable data is referred to as the *cache consistency* or *cache coherency* problem. Inconsistencies arise when two or more processors have copies of the same shared memory location in their private caches, and one processor modifies the location but fails to communicate the change to the other processors.

Cache consistency algorithms prevent the old data, called *stale data*, from being used. The two approaches traditionally used are (1) to update main memory and cause cache invalidations on each write, or (2) to use software assists. The first approach, called *write-through with invalidation*, generates bus traffic proportional to the number of writes. This seriously degrades performance in a system with several high-performance processors.[17] The second approach requires software to identify whether data is potentially shared and makes use of noncacheable pages or write-through with invalidation to keep that data consistent. This generates more bus traffic than our approach for unrestricted sharing, because bus transactions are generated on many references to shared pages even if most of the data is not in simultaneous use.

Other researchers are currently investigating how to improve the effectiveness of the software approach by using synchronization primitives to delay the invalidation of stale data.[18,19] The principal weakness of the software approach is that it may require extra effort from the programmer, possibly discouraging the development of parallel processing software.

The cache consistency algorithm used in SPUR, called *Berkeley Ownership*, is based on the concept of ownership of cache blocks.[20] The responsibility for maintaining consistency is distributed among the caches. If a cache owns a block, then no copies of the block occur in any other caches. The owner may update the cached entry locally without broadcasting its actions. If a cache does not own a block, it must first obtain ownership before it can update the block. Ownership is obtained by a broadcast to other caches, causing them to invalidate their copies of the block. In addition to the local update privilege, ownership carries the obligation to update main memory on block replacement (copy-back) and the responsibility of overriding main memory if another cache requests the block.

SPUR implements Berkeley Ownership with standard memory, a dual-address bus, and snooping caches. The bus broadcasts ownership requests and transfers cache blocks. Most bus transactions begin with a type field (such as read or read-for-ownership) and a block address (both virtual and physical), and end with a data transfer.[21]

Each processor cache controller is supplemented with hardware, called the *snoop*, that monitors the bus for transactions involving blocks that it has cached. The snoop compares the virtual addresses of all bus transactions with a second copy of the cache's address tags. If a match occurs, the snoop may have to invalidate its copy of the block or override main memory and provide the data to complete the bus transaction. The latter action only occurs for blocks that have been modified and are simultaneously shared by processes on more

than one processor. While we have little data on sharing, we expect this to occur on a small fraction of all transactions.

Berkeley Ownership, implemented in hardware with snooping caches, serves the goals of SPUR in several ways. First, it preserves the shared memory model. This model facilitates parallel processing experiments by providing a simple, flexible mechanism for sharing data between processes. Second, it simplifies parallel processing software by relieving programmers of the responsibility of understanding shared caches. Third, the Berkeley Ownership protocol has good multiprocessor performance because it can be restricted to generate extra bus transactions only when two or more processors are simultaneously accessing writable shared data. For example, our protocol allows semaphores to be cached. No bus traffic is needed to modify a semaphore if only one process happens to be using the semaphore for some period of time, or if all the processes using the semaphore are on the same processor. Other methods generate bus transactions after shared data has been modified even if no processes on other processors are trying to access the same data. Fourth, our protocol yields good uniprocessor performance. When no interprocessor sharing can occur, no consistency-preserving bus transactions will be made. Fifth, the algorithm is not difficult to implement. It requires an additional state machine in the cache controller, two additional state bits for each 32-byte cache block, a second copy of all cache state bits and address tags, and a change to the system bus to permit snooping. It does *not* require centralized control or any memory board modifications.

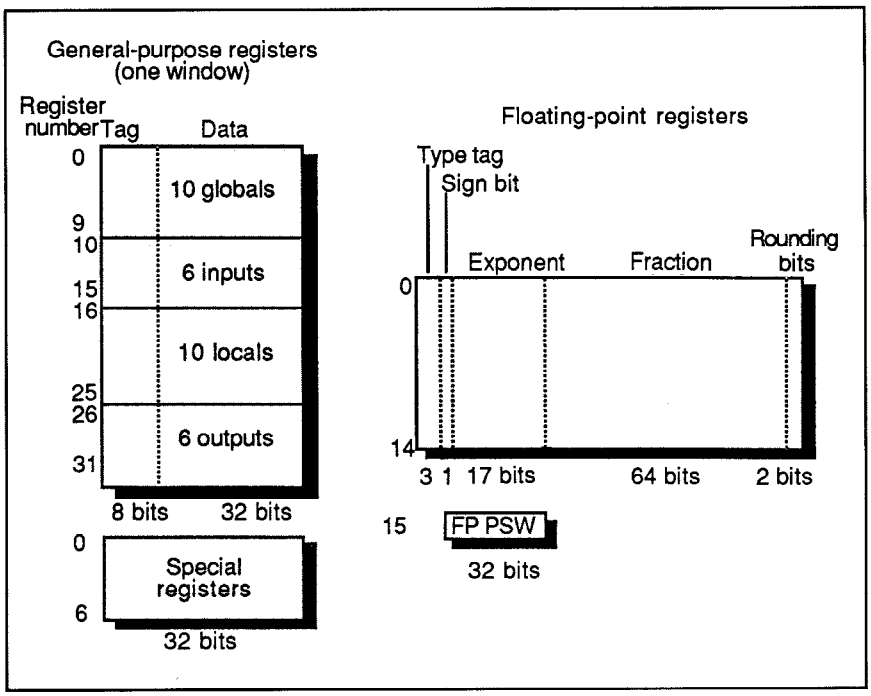


**Figure 5. SPUR registers. SPUR's registers divide into three groups: general purpose, special, and floating point. The general-purpose registers are organized into fixed-size overlapping windows so that the output registers of one window become the input registers of the next window after a procedure call. Only one 32-register window is visible at a time. The entire general-purpose register file contains eight windows (not shown) for a total of 138 registers. The general-purpose registers are 40 bits wide, consisting of an 8-bit tag and 32 bits of data. The special registers include the user and kernel processor status words, register window pointers, and several program counters. The floating-point registers are 87 bits wide to accommodate SPUR's representation of IEEE extended-precision numbers. The representation includes a three-bit type tag to simplify detection of infrequent floating-point types (such as Not-a-Number). The 15 floating-point registers and the floating-point processor status word (a special register) are implemented on the FPU chip rather than on the CPU chip to improve operand access time for floating-point instructions. Multiple windows of these registers were not implemented because of insufficient FPU chip area.**

## The CPU and floating-point coprocessor

The SPUR CPU design evolved from the RISC II design.[8] Like RISC II, SPUR has a streamlined instruction set and a large register file with multiple, overlapping register windows to speed up procedure calls. For several reasons, the instruction set is well-suited for a high-performance VLSI implementation without microcode. First, the instructions are easy to decode because of their fixed size and few formats. Second, computational instructions operate exclusively on registers, while memory can be accessed only with load and store instructions. Register-to-register instructions execute quickly and deterministically because they cannot generate cache misses or page faults once they begin execution. Third, the instructions perform simple operations implemented in a short, uniform pipeline. Every instruction uses a particular resource in the same pipeline stage. For example, all SPUR instructions use the arithmetic and logic unit (ALU) to combine operands or calculate an effective address in the second stage of the pipeline. This simplifies the hardware by predetermining the scheduling of resources.

The differences between SPUR and RISC II are products of technological improvements and the new goals of supporting Lisp and floating-point arithmetic. Technological improvements in the past few years have increased the number and speed of transistors possible on a VLSI chip. In SPUR, the additional transistors are used in an on-chip instruction cache, for tagging Lisp data, and in a low-overhead interface to a floating-point coprocessor.

### General-purpose features

*The register set.* The SPUR register set, shown in Figure 5, includes 32 general-purpose registers. Like the RISC II chip, the SPUR CPU contains several copies of the general-purpose register set

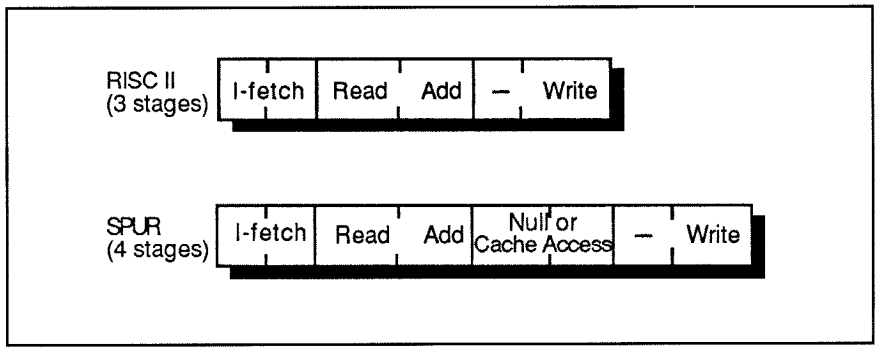


**Figure 6. RISC II and SPUR pipelines. RISC II used a three-stage pipeline (top) that required the pipeline to stall one cycle on every data memory reference so that precisely one memory reference (instruction fetch or data access) was done every cycle. The first stage fetched the next instruction from memory; the second read two registers and performed an ALU operation; the final stage wrote the result into a register.**

**SPUR uses a four-stage pipeline (bottom) so that an instruction can be issued every cycle. Memory-accessing instructions use the additional stage to do a cache access. Register-to-register instructions do nothing in the additional stage. All instructions write the register file in the fourth stage to guarantee that no two instructions try to write at the same time. Pipeline hardware guarantees that the result of a register-to-register instruction can be used by the subsequent instruction even though the result has not yet been written into the register file. However, software must guarantee that the result of a load instruction is not used by the instruction that (dynamically) follows the load.**

(not shown in Figure 5) so that these registers do not have to be saved in memory and restored on most procedure calls and returns. In addition, the register windows for a caller and a callee overlap by six registers so that most arguments and returned values can be passed in-place in registers instead of in memory. For both reasons, overlapped register windows reduce the time required for procedure calls and returns. The primary cost of the multiple register sets is a significant amount of chip area and, to a lesser extent, slower register access time and increased process switching overhead.

*The execution pipeline.* The SPUR execution pipeline is one stage longer than the three-stage RISC II pipeline (see Figure 6). RISC II could issue a register-to-register instruction every cycle. It used resources efficiently: in every cycle two registers were read, one was written, the ALU was utilized, and the path to memory was used to fetch an instruction. Unfortunately, this arrangement left no memory bandwidth for data references. Consequently, loads and stores had to stall the pipeline one cycle to use the path to memory. Thus, RISC II did a memory reference per cycle rather than completing an instruction per cycle.

SPUR uses an instruction buffer and a four-stage pipeline to attempt to issue and complete an instruction every cycle. The instruction buffer satisfies most instruction fetches without cache accesses. The new pipeline stage allows memory referencing instructions to make cache accesses without stalling the pipeline, but forces register-to-register instructions to delay their register write for one stage. All instructions modify the general-purpose register file in the fourth pipeline stage, thereby avoiding write conflicts. Internal forwarding is done by the hardware so that the result of a register-to-register instruction can be used by the next instruction even though that result has not yet been written into the register file.

In practice, SPUR will not be able to execute one instruction every cycle, principally because of instruction buffer and cache misses. On simulations with the Gabriel benchmarks (see Table 1), SPUR executed an instruction every 1.59 cycles with instruction buffer and cache miss ratios of 14 and 1 percent. Performance varied across the benchmarks from 1.03 to 1.99 cycles per instruction. Larger programs are likely to have more cycles per instruction because of poorer locality of reference. Even if the instruction buffer and cache miss ratio double, however, SPUR still executes an instruction every 2.06 cycles.

*The instruction set.* This section focuses on a few important decisions in the SPUR instruction set. (See Taylor's "SPUR Instruction Set Architecture"[22] for the complete design.) To simplify decoding, all instructions are four bytes long and use fixed positions for the opcode and register specifiers. Most instructions use either two source registers and one destination register or a source register, an immediate constant, and a destination register. Table 3 lists the basic instruction set, not including instructions for Lisp and floating-point operations.

Memory accesses are made with loads and stores. The effective address for a load is either the sum of two registers or the sum of an immediate displacement and a register. SPUR uses a *delayed* load, which requires software to guarantee that the result of a load instruction is not used by the instruction that (dynamically) follows the load. Cache misses on data references stall the entire pipeline and thus are not visible to software. The effective address for a store is always a register plus immediate displacement, so that a two-port register file suffices (one register for the address and one for the data). A store stalls the execution pipeline for one cycle, because less-common cache writes take longer than cache reads.

Cache reads access cache data in parallel with examining cache address tags. Cache writes begin in a similar fashion, but cannot write into a cache block until after the address tag has been examined. In our initial design, stores did not stall the pipeline because we set the cycle time to the cache write time. We were able to improve performance by reducing the cycle time to the cache read time, thus forcing the less frequent cache writes to take two cycles.

SPUR supports synchronization with a test-and-set instruction implemented in the cache. Under the best of conditions it does not require any bus transactions. To simplify the cache interface, SPUR does not have load or store instructions that manipulate individual bytes. A load-byte instruction would increase the cache access time, and a store-byte instruction would increase cache complexity. Instead, byte insert and extract instructions assist in loading and storing individual bytes.

SPUR adopted the delayed branch from RISC II. The execution of a branch instruction on most pipelined processors requires that the branch target be fetched and the execution pipeline flushed before the target instruction is executed. A branch instruction on SPUR allows—in fact, requires—the next sequential instruction to be executed while the branch target is fetched. A delayed branch saves program execution time if a useful instruction can be scheduled in this *delay slot*. Gross found this could be done on 63 percent of delayed branches dynamically encountered in the traces studied.[23] Gross also found that delayed branches did not significantly increase code size since 87 percent of the statically examined delayed slots contained useful instructions.

SPUR also includes a *canceling* compare and branch instruction, which dynamically turns the instruction in the delay slot into a no-op if the branch is not taken. The technique is also being used in the Lawrence Livermore S-1 AAP. This variant of the delayed branch makes it

**Table 3. Basic SPUR instructions.**

This table lists the basic SPUR instruction set. The column Cycles shows the minimum number of cycles consumed by an instruction. Many instructions operate on two sources (src1 and ri) and write a result into a destination (dest). Src1 and dest are five-bit register specifiers. Ri is either a 5-bit register specifier or a 14-bit signed immediate constant. Rci stands for a five-bit register specifier or a five-bit unsigned immediate constant. Pc stands for the program counter. The Action column describes what happens in the data portion of the destination and source registers. Exceptional conditions and Lisp tag manipulation are described in the SPUR instruction set architecture.[23]

| Instruction | Operands | Action | Cycles |
|---|---|---|---|
| Load/Store | | | |
| load_32 | dest, src1, ri | dest ← M [src1 + ri] | 1 |
| load_external | dest, src1, ri | dest ← external (cache) state | 1 |
| test_and_set | dest, src1, ri | dest ← M [src1 + ri]; M [src1 + ri]<00> ← 1 | 2 |
| store_32 | src2, src1, imm | src2 → M [src1 + imm] | 2 |
| store_external | src2, src1, imm | src2 → external (cache) state | 1 |
| Compute | | | |
| add, subtract | dest, src1, ri | dest ← src1 op ri | 1 |
| add(no traps) | dest, src1, ri | dest ← src1 + ri | 1 |
| and, or, xor | dest, src1, ri | dest ← src1 op ri | 1 |
| sll, srl, sra | dest, src1, ri | dest ← src1 op ri<01:00> | 1 |
| extract | dest, src1, ri | dest<07:00> ← one byte from src1 selected by ri | 1 |
| insert | dest, src1, ri | dest ← ri<07:00> inserted into one byte of src1 | 1 |
| Branch/Jump | | | |
| cmp_branch_delayed | cond, src1, rci, offset | if (src1 cond rci) pc ← pc + signed word offset | 1 |
| cmp_branch_likely | cond, src1, rci, offset | if (src1 cond rci) pc ← pc + signed word offset<br>else change next instruction into no-op | 1 |
| jump | address | pc ← word address (in same segment) | 1 |
| jump_register | src1, ri | pc ← src1 + ri | 1 |
| Call/Return | | | |
| call, call_kernel | address | increment current window pointer;<br>save pc; pc ← word address | 1 |
| return, return_trap | src1, ri | pc ← src1 + ri<br>decrement current window pointer | 1 |
| Access Specials | | | |
| read_special | dest, src1 | dest ← special register src1 | 1 |
| write_special | dest, src1, ri | special register dest ← src1 + ri | 1 |
| read_kernel_psw | dest | dest ← kernel psw | 1 |
| write_kernel_psw | src1, ri | kernel psw ← src1 + ri | 1 |

**Table 4. SPUR Lisp instructions.**

This table lists Lisp instructions. The column Cycles shows the minimum number of cycles consumed by an instruction. Load_40 and store_40 move tagged words into and out of registers. Car and cdr are special forms of load_40 that check for a proper list element. Read_tag and write_tag move a tag to and from the data part of a register. Compare_and_branch_delayed and compare_and_branch_likely, presented in Table 3, compare the tags and values of two Lisp data items. In addition, tag_compare_and_branch_delayed and tag_compare_and_branch_likely are available to determine the value of a tag (by comparing it with an immediate constant). Compare_and_trap and tag_compare_and_trap are used to test for error conditions.

| Instruction | Operands | Action | Cycles |
|---|---|---|---|
| load_40 | dest, src1, ri | dest ← M [src1 + ri] | 1 |
| car/cdr | dest, src1, ri | dest ← M [src1 + ri] | 1 |
| store_40 | src2, src1, imm | src2 → M [src1 + imm] | 2 |
| read_tag | dest, src1 | dest < 07:00 > ← src1 tag | 1 |
| write_tag | dest, ri | dest tag ← ri < 07:00 > | 1 |
| tag_cmp_branch_delayed | cond, src1, tag_imm, offset | if (src1 < tag > cond tag_imm)<br>pc ← pc + signed word offset | 1 |
| tag_cmp_branch_likely | cond, src1, tag_imm, offset | if (src1 < tag > cond tag_imm)<br>pc ← pc + signed word offset<br>else change next instruction into no-op | 1 |
| compare_and_trap | cond, src1, rci | if (src1 cond rci) trap | 1 |
| tag_compare_and_trap | cond, src1, tag_imm | if (src1 cond rci) trap | 1 |

easier to schedule a useful instruction in the delay slot. The natural use of this instruction is at the bottom of a loop, with the branch target set to the loop's second instruction and the delay slot filled with a copy of the loop's first instruction.

An arbitrary shift instruction was not included, because most shifting done in high-level language programs is for effective address computation in arrays and records.[8] SPUR provides shift instructions only to shift one bit right and one, two, and three bits left. Shift operations are not needed for integer multiplication or division since these operations are done with the FPU.

**Supporting Lisp.** The Lisp programming language has some features difficult to implement efficiently on conventional computers. These include frequent function calls and returns, polymorphic operations, and automatic garbage collection. Most machines designed to run Lisp use a stack-based architecture with extensive microcode support (such as the Symbolics 3600,[24] LMI Lambda,[25] and the Xerox D-Machines.[26]) Our approach emphasizes a simple, regular instruction set, overlapping register windows, and tagged data.[9] Table 4 lists the instructions tailored for Lisp.

Fast function calls and returns are particularly important for Lisp, because Lisp programs are constructed from many small functions. SPUR provides fast function calls and returns through the overlapping register window mechanism. Studies have shown that this mechanism, developed for C, effectively speeds up Lisp calls and returns.[27] The complicated argument options allowed by Common Lisp (such as default and keyword parameters) are handled by software rather than by special-purpose instructions or microcode. This approach increases the size of functions that use these options, but ensures that simple function calls execute rapidly.

*Tagged architecture.* Lisp uses *polymorphic* functions with operands whose type is not known until run-time. A polymorphic function operates on arguments of more than one data type.[28] For example, the addition operator (+) is a polymorphic operator in most high-level languages because it is defined to operate on both integers and floating-point numbers. Lisp complicates the implementation of polymorphic operations because it associates the type of data with the data values instead of the program variables. For example, a variable is not an integer variable, known at compile-time, but rather a variable that may contain an integer at run-time. When a Lisp function is evaluated, the types of operands must be determined before the appropriate routine is executed.

SPUR handles polymorphic operations by manipulating the 6-bit data-type tags of operands in parallel with operating on the 32-bit data values (see Figure 7). Type checking in SPUR, like several other machines,[29,3] assumes that most arithmetic operands are integers. For example, a polymorphic add operation in SPUR is implemented with an add instruction that begins by adding the 32-bit operands as if they were integers and, in parallel, checking the data-type tags to verify that they are integers. If both operands are integers, the instruction finishes by writing the sum into the result register. Otherwise, the register write is suppressed and the instruction traps to software that determines the types of the operands and performs the appropriate form of addition.

The power of SPUR to manipulate data-type tags is increased by several instructions that allow conditional traps and branches based on tag values (see Table 4). The conditional traps allow efficient checking of error conditions. Explicit tag comparison instructions are used to implement polymorphic operations in the more complicated cases not handled by the hardware.

Data-type tags also assist list manipulation, which is fundamental in Lisp. A list is a sequence of elements (such as (*a b c*)). The Lisp functions that manipulate lists are called *car* and *cdr*. Car returns

the first element of a list ($a$) and cdr returns the rest of the list ($(b\ c)$).

Car and cdr can be implemented with load instructions because lists are stored as linked lists in main memory. However, the semantics of Common Lisp strongly encourage generation of an exception if the argument of car or cdr is not a list. Conventional architectures must execute one or more instructions to check this condition even though the arguments of all car's and cdr's in a correct program are lists.

SPUR provides a car/cdr instruction that checks the data-type tag in parallel with the load. A trap is generated if the type of the operand is inappropriate. This is an ideal use of parallel tag checking because it allows SPUR to execute car and cdr at the same speed as a load and still generate exceptions on errors.

SPUR also uses part of the tag field to assist in garbage collection. Lisp encourages programmers to dynamically create and use data structures in memory. Automatic garbage collection reclaims structures that are no longer in use. This feature relieves the Lisp programmer of the responsibility of explicitly discarding obsolete structures, a task that leads to subtle bugs and complicated programming. SPUR stores a two-bit *generation number* in the tag to assist a *generation scavenging* garbage collection algorithm (see Figure 7).[3] The algorithm exploits a property of dynamic data: new data structures are likely to become garbage soon and old data structures are likely to stay in use. Therefore, most garbage collection activity focuses on the new data. The generation number records the number of garbage collections that an item has survived and hence its age.

*Poor data density.* We designed the SPUR architecture with more emphasis on speed and simplicity than concern for code or data density. The prototype implementation has particularly poor Lisp data density because we decided not to build a complete 40-bit system.

The CPU manipulates 40-bit data (an 8-bit tag and 32-bit data). That data must often be loaded from and stored to the cache and the rest of the memory system. Three approaches exist for doing this:

(1) build the whole system with 40-bit words,

(2) allow unaligned cache accesses, or

(3) place 40-bit words in aligned 64-bit words.

We rejected a 40-bit-word memory system because it would preclude the use of many off-the-shelf subsystems, which would substantially delay completion of the prototype. It would also have complicated non-Lisp software in such areas as string manipulation and file transfer with non-SPUR machines. We rejected unaligned cache accesses because of the complexity they would add to the cache. An unaligned access can cross a cache block boundary, possibly forcing the cache to handle two cache misses and the associated address translation. Consequently, we chose to store 40-bit Lisp words in aligned 64-bit words. The other 24 bits are wasted for tagged Lisp data, but not for instructions, data for other languages, and some Lisp data structures. At worst, this storage strategy uses 60 percent more Lisp data memory than the first two schemes, but it allows us to explore ideas more quickly by simplifying the design of the prototype.

**Floating-point support.** SPUR implements the IEEE 754 binary floating-point standard[5] with a mixture of hardware and software. Floating-point instructions are executed on the floating-point coprocessor chip (FPU). The FPU hardware is optimized to execute common floating-point operations quickly. Effective use of the FPU depends on a low-overhead floating-point interface and support for concurrent execution of floating-point and CPU instructions.

The SPUR FPU is one of the first implementations of IEEE floating-point that does not use any microcoded control. The Fairchild Clipper CPU also has a hard-wired IEEE floating-point unit.

*The floating-point coprocessor.* Floating-point instructions are either register-to-register instructions or loads and stores (see Table 5). The register-to-register instructions include add, subtract, multiply, and divide. Except for multiply (7 cycles) and divide (19 cycles), a new floating-point instruction can be issued every four cycles.

Data are transferred between the FPU and the cache with floating-point load and store instructions. Floating-point load instructions convert all single (32 bits) and double (64 bits) precision numbers to extended precision to simplify the computational instructions. A convert instruction must be executed before a store to perform the inverse operation. For example, an extended_to_single convert instruction must be executed before a store_single instruction.

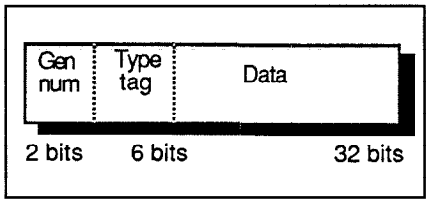


**Figure 7. Lisp tagged data. SPUR augments Lisp data words with an eight-bit tag that includes a six-bit data-type tag and two-bit generation number. Lisp integers and characters are represented as immediate data. All other types of Lisp objects are referenced by typed pointers. Some of the tag values are used by the hardware to do tag checks in parallel with data operations. Other tag values are interpreted only by software. The generation number is used to implement a generation-scavenging garbage-collection algorithm.**

The FPU contains 15 87-bit floating-point registers organized as a single register set (see Figure 5). There is no analog to the overlapping windows used for the general-purpose registers because of insufficient FPU chip area to implement more registers. Furthermore, more research is needed to determine how to use overlapping windows for floating-point registers.

The floating-point register set is independent of the general-purpose register set for four reasons:

(1) to reduce access time for floating-point operands,
(2) to allow more freedom in setting the width of floating-point registers,
(3) to permit concurrent execution of integer and floating-point operations, and
(4) to permit implementation of a separate FPU chip.

SPUR divides the floating-point standard into two parts. One part is implemented by a set of instructions (see Table 5) with hard-wired control and the other is implemented by software trap handlers. The standard defines six types of floating-point numbers: zeros, normalized numbers, denormalized numbers, infinities, and two types of Not-a-Number symbols. The FPU manipulates normalized numbers and zeros entirely in hardware. The other four less-common types require software assistance.

The FPU manipulates single (32 bits),

**Table 5. SPUR floating-point instructions.**

This table lists SPUR floating-point instructions. The column Cycles shows the minimum number of cycles consumed by an instruction in normal operating mode. If the FPU and CPU are operated concurrently, a CPU instruction can begin one cycle after an FPU instruction has started (see the section labeled Floating-point coprocessor interface). There are floating-point load and store instructions for each floating-point format and for integers. Extended-precision numbers require two different loads and two different stores to move the first 64 bits and the last 64 bits. Loads do implicit conversion to extended precision, but stores merely copy bits. Store_single, store_double, and store_integer must be preceded by the corresponding convert instruction (such as extended_to_ single). The to_cpu and from_cpu instructions transfer integers directly between the integer and floating-point register sets so that the FPU can be used effectively for integer multiply and divide. A conditional branch based on floating-point data is done in two steps. First, the CPU executes an fcmp instruction to set a bit in the floating-point processor status word (FP PSW). Second, the CPU executes a conditional branch instruction that tests this bit. Most floating-point operations execute in four cycles using the add/subtract hardware. Multiply and divide use additional special-purpose hardware. A sync instruction is used when the CPU and FPU are executing instructions in parallel. It forces the CPU to stop executing new instructions until the FPU completes its current instruction (if any). This can be used to guarantee that the store of a floating-point result does not begin before the result has been computed.

| Instruction | Operands | Action | Cycles |
|---|---|---|---|
| load_single | dest, src1, ri | FPU dest ← (convert to extended) M [src1 + ri] | 1 |
| load_double | dest, src1, ri | FPU dest ← (convert to extended) M [src1 + ri] | 1 |
| load_extended1 | dest, src1, ri | FPU dest ← M [src1 + ri] | 1 |
| load_extended2 | dest, src1, ri | FPU dest ← M [src1 + ri] | 1 |
| load_integer | dest, scr1, ri | FPU dest <63:32> ← M [src1 + ri] | 1 |
| store_single | src2, src1, i | FPU src2 → M [src1 + i] | 2 |
| store_double | src2, src1, i | FPU src2 → M [src1 + i] | 2 |
| store_extended1 | src2, src1, i | FPU src2 → M [src1 + i] | 2 |
| store_extended2 | src2, src1, i | FPU src2 → M [src1 + i] | 2 |
| store_integer | src2, src1, i | FPU src2 → M [src1 + i] | 2 |
| from_fpu | dest, src2 | CPU dest ← FPU src2 <63:32> | 1 |
| to_fpu | dest, src2 | FPU dest <63:32> ← CPU src2 | 1 |
| fadd, fsub | dest, src1,src2 | FPU dest ← FPU src1 op FPU src2 | 4 |
| fmul | dest, src1, src2 | FPU dest ← FPU src1* FPU src2 | 7 |
| fdiv | dest, src1, src2 | FPU dest ← FPU src1/FPU src2 | 19 |
| fcmp | src1, src2 | FP_PSW <branch_bit> ← (FPU src1 cond FPU src2) | 4 |
| fnegate | dest, src1 | FPU dest ← FPU src1 with opposite sign | 4 |
| fabs | dest, src1 | FPU dest ← FPU src1 with positive sign | 4 |
| fmov | dest, src1 | FPU dest ← FPU src1 | 4 |
| int_to_extended | dest, scr1 | FPU dest ← (convert to extended) FPU src1 <63:32> | 4 |
| extended_to_int | dest, src1 | FPU dest <63:32> ← (convert to integer) FPU src1 | 4 |
| extended_to_single | dest, src1 | FPU dest ← (convert to single) FPU src1 | 4 |
| extended_to_double | dest, src1 | FPU dest ← (convert to double) FPU src1 | 4 |
| sync | | CPU waits until FPU is not busy | 1 |

double (64 bits), and extended-precision numbers (at least 79 bits) in a common 87-bit format to reduce hardware complexity. SPUR enlarges the minimum extended-precision format in four ways.

First, a three-bit tag identifies the type of a number. This tag reduces the time needed by a load instruction to convert numbers to extended-precision by allowing the load to handle exponents for all types of numbers in a uniform fashion. In addition, the hardware for computational instructions can determine whether software assistance is necessary by examining the three-bit tag rather than the entire number.

Second, SPUR expands the exponent by two bits so that trap handlers can adjust denormalized operands. This enables SPUR to multiply and divide denormalized numbers using hardware designed for normalized operands.

Third, two rounding bits are added so that SPUR can mimic rounding from an infinitely-precise result to a precision shorter than extended precision. This feature is necessary to correctly handle a denormalized number produced by an underflow exception.

Fourth, one bit is used to hold the most significant fraction bit in explicit form.

*The floating-point coprocessor interface.* The FPU is sufficiently fast that the performance of floating-point operations is sensitive to the overhead associated with starting floating-point operations and the overhead of transferring floating-point operands to and from the FPU. Consequently, 28 CPU pins are

used to implement a low-overhead interface between the CPU and the FPU. Unfortunately, the close coupling of the two chips may make it difficult to use the SPUR FPU without the SPUR CPU.

To reduce the overhead of starting floating-point operations, the FPU tracks all CPU instructions using 22 pins dedicated to carrying opcode, register specifiers, and other control information to the FPU (and possibly other coprocessors). Some commercial floating-point coprocessors track instructions by monitoring CPU instruction fetches to memory (such as the Intel 8087). However, this will not work in SPUR because the CPU fetches most instructions from the on-chip instruction buffer.

The SPUR floating-point interface reduces operand overhead in three ways. First, the floating-point registers reside on the FPU. Since all floating-point computation instructions operate with operands in these registers, intermediate results can be efficiently used.

Second, floating-point load and store instructions transfer data directly between the FPU and the cache. In contrast, many commercially available interfaces require floating-point data to be transferred through the CPU. The following sequence occurs when a floating-point load instruction is issued by the CPU: the FPU recognizes the floating-point load instruction and saves the destination register specifier; the CPU calculates the effective memory address and sends the address to the cache; the cache sends the data to both the FPU and the CPU; the FPU reads the data and loads it into the appropriate floating-point register; the CPU ignores the data, but recognizes that the load is complete.

Third, the data path between the FPU and the memory system is 64 bits wide, in constrast to more commonly used 8-, 16-, and 32-bit interfaces. This allows load and store instructions to move single and double-precision numbers with a single transfer and extended-precision numbers with two transfers. SPUR's wide FPU interface reduces the probability that operand movement will limit floating-point throughput, which can easily occur for double-precision computations.

The coprocessor interface also allows concurrent CPU and FPU operation. Subject to some software constraints, the CPU can continue executing general-purpose instructions, Lisp instructions, and floating-point loads and stores while the FPU is busy. Overlapping operand movement/index calculations with floating-point operations can halve the execution time of many inner loops of floating-point intensive programs.[30] However, software must restrict the interaction between concurrently executed instructions by reordering instructions or by inserting *sync* instructions. For example, a sync instruction must be inserted between a floating-point operation and an instruction that stores the result of the operation in memory if the store could read the result register before it is written.

## Status

The implementation of SPUR is in progress. As of September 1986 the custom components and processor board have been described at the register-transfer level with a variant of the ISP language and simulated with a software package called N.2. The layouts of the custom chips are near completion. They all use four-phase nonoverlapping clocks with a projected cycle time of 100 to 150 nanoseconds. The processor board has been designed, simulated with N.2, and is nearly ready for physical implementation. We expect to have working components by early 1987 and a working system later in the year.

SPUR is a multiprocessor research vehicle, but we have not as yet been able to run multiprocessing experiments. Nevertheless, we have some preliminary results.

First, selected architectural changes can significantly ease implementation and, at the same time, improve performance. For example, disallowing synonyms enabled us to build virtually-tagged caches without complex reverse-translation mechanisms. Virtually-tagged caches improved performance by reducing cache access time and permitting slow address translation.

Second, in-cache address translation keeps PTEs consistent and offers performance comparable to a translation buffer at less cost.

Third, cache consistency can be maintained in hardware at reasonable cost and without any modifications to main memory boards.

Fourth, Lisp can be supported without a stack-based architecture and without a microcoded implementation. However, data-type tags or some other direct support of Lisp's dynamically-typed data are advantageous.

Fifth, IEEE standard floating-point can be implemented without microcoded control if software handles the less common cases.

Sixth, floating-point coprocessor interfaces can be designed to significantly reduce operand-movement overhead by putting the floating-point registers on the floating-point coprocessor and loading these registers directly from a cache using a 64-bit data path.

The goal of the first phase of the SPUR project is to design and implement several working prototypes. If the prototypes meet our expectations, we hope to find partners to help us transfer SPUR from academia to industry. □

## Acknowledgments

The SPUR project is a cooperative project that benefits from the contributions of many people within the Berkeley community besides the authors of this paper. The implementation of the CPU, FPU, and CC was begun by class members of CS 292I, taught in Spring 1985 by Randy Katz. Members of this class who assisted included Chien Chen, Li-fan Pei, Rick Rudell, Trudy Stetzler, Sinohe Villalpando, Albert Wang, Don Webber, and Tom Wisdom. The implementation of three VLSI chips would not have been possible without computer-aided design software developed by Gordon Hamachi, Bob Mayo, John Ousterhout, Walter Scott, and George Taylor. The architecture of SPUR has been strongly affected by interactions with the SPUR operating systems group, consisting of Andrew Cherenson, Fred Douglis, John Ousterhout, Mike Nelson, and Brent Welch.

We would also like to thank Sue Dentinger, Dave Ditzel, Gregg Foster, Jim Goodman, Robert Henry, Louis Monier, Prabhakar Ragde, Dick Sites, Alan Smith, Jim Smith, Chuck Thacker, and John Wakerly for their suggestions, which improved the quality of this paper.

SPUR was first presented at the 1985 Asilomar Microcomputer Workshop.

Principal funding for the SPUR project is provided by the Defense Advanced

Research Projects Agency under contract N00039-85-C-0269. Additional support for this research was provided by the State of California MICRO program, by a Digital Equipment Corporation CAD/CAM grant, by the National Science Foundation under grant DCR-8202591, by equipment donations from Texas Instruments, Inc., and by computer resources provided under DARPA contract N00039-84-C-0089.

## References


1. J. Deminet, "Experience with Multiprocessor Algorithms," *IEEE Trans. on Computers,* Vol. C-31, No. 4, Apr. 1982.
2. D. A. Patterson and C. H. Sequin, "A VLSI RISC Computer," Vol. 15, No. 9, Sept. 1982, pp. 8-21.
3. D. Ungar et al., "Architecture of SOAR: Smalltalk on a RISC," *Proc. 11th Int'l Symp. on Computer Architecture,* June 1984, p. 1887.
4. G. L. Steele, "Common Lisp: The Language," Digital Press, Burlington, MA, 1984.
5. IEEE Standard 754-1985 for Binary Floating-Point Arithmetic, 1985, order number CN953.
6. R. Olson, "Parallel Processing in a Message-Based Operating System," *Software,* July 1985, pp. 39-49.
7. D. A. Patterson, "Reduced Instruction Set Computers," *CACM,* Jan. 1985, pp. 8-21.
8. M. G. H. Katevenis, "Reduced Instruction Set Computer Architectures for VLSI," PhD thesis, UC Berkeley, Oct. 1983.
9. G. S. Taylor et al., "Evaluation of the SPUR Lisp Architecture," *Proc. 13th Int'l Symp. on Computer Architecture,* Tokyo, Japan, June 1986.
10. R. P. Gabriel, *Performance and Evaluation of Lisp Systems,* MIT Press, 1985.
11. B. W. Lampson and D. D. Redell, "Experience with Processes and Monitors in Mesa," *CACM,* Feb. 1980.
12. J. R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," *Proc. 10th Int'l Symp. on Computer Architecture,* Stockholm, Sweden, June 1983, pp. 124-131.
13. M. D. Hill and A. J. Smith, "Experimental Evaluation of On-Chip Microprocessor Cache Memories," *Proc. 11th Int'l Symp. on Computer Architecture,* Ann Arbor, MI, June 1984.
14. R. H. Katź et al., "Memory Hierarchy Aspects of a Multiprocessor RISC: Cache and Bus Analyses," CS Div. tech. report UCB/CSD 85/221, UC Berkeley, Jan. 1985.
15. A. J. Smith, "Cache Memories," *Computing Surveys,* Vol. 14, No. 3, Sept. 1982, pp. 473-530.
16. P. J. Denning, "Virtual Memory," *Computing Surveys,* Vol. 2, No. 3, Sept. 1970.
17. D. A. Wood et al., "An In-Cache Address Translation Mechanism," *Proc. 13th Int'l Symp. on Computer Architecture,* Tokyo, Japan, June 1986.
18. L. M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *Trans. on Computers,* Vol. C-27, No. 12, Dec. 1978, pp. 1112-1118.
19. *Biannual Research Summary, DARPA VLSI Contracts,* Stanford Univ. CSL and ICL.2 book, Stanford, Nov. 1984-Mar. 1985.
20. A. J. Smith, "CPU Cache Consistency with Software Support Using One-Time Identifiers," *Proc. Pacific Computer Comm. Symp.,* Seoul, Republic of Korea, Oct. 1985.
21. R. H. Katz et al., "Implementing a Cache Consistency Protocol," *Proc. 12th Int'l Symp. on Computer Architecture,* Boston, MA, June 1985.
22. G. Gibson, "SPURBUS Specification," *Proc. of CS292i: Implementation of VLSI Systems,* ed. R. H. Katz, UC Berkeley, Sept. 1985. Also Computer Science Div. tech. report UCB/CSD 86/259.
23. G. Taylor, "SPUR Instruction Set Architecture," *Proc. of CS292i: Implementation of VLSI Systems,* ed. R. H. Katz, UC Berkeley, Sept. 1985. Also Computer Science Div. tech. report UCB/CSD 86/259.
24. T. Gross, "Code Optimization of Pipeline Constraints," PhD thesis, Stanford Univ., Aug. 1983.
25. D. A. Moon, "Architecture of the Symbolics 3600," *Proc. 12th Symp. on Computer Architecture,* Boston, MA, June 1985.
26. "The Lambda System: Technical Summary LMI," Lisp Machine, Inc., 1983.
27. R. R. Burton et al., *Papers on Interlisp-D,* Xerox PARC tech. report SSL-80-4, Sept. 1980.
28. C. Ponder, "But Will RISC Run Lisp? (A Feasibility Study)," unpublished masters report, UC Berkeley, Apr. 1983.
29. J. Donahue and A. Demers, "Data Types Are Values," *ACM Trans. on Programming Languages and Systems,* July 1985.
30. C. B. Roads, "3600 Technical Summary," Symbolics, Inc., Cambridge, MA, 1983.
31. P. M. Hansen, "Coprocessor Architectures for VLSI," unpublished thesis research proposal, UC Berkeley, May 1985.

# SPUR
# Research Group

## University of California, Berkeley