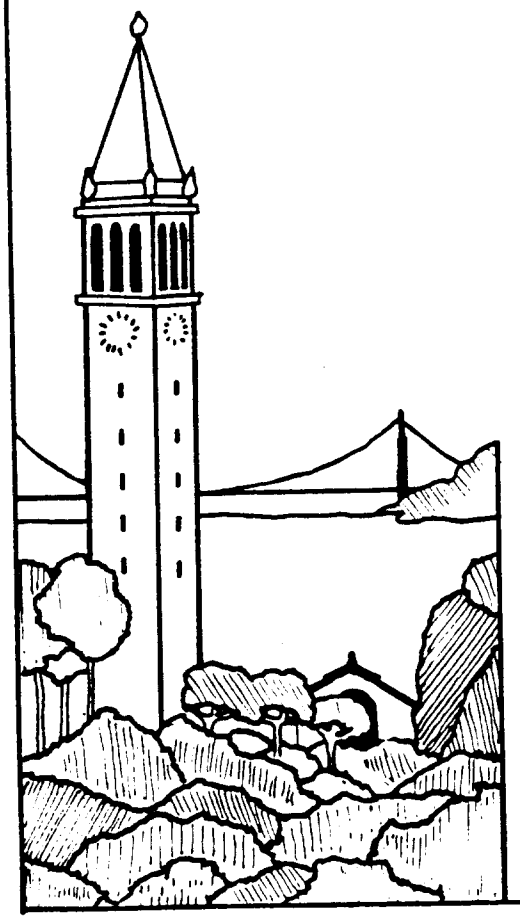
SPUR: A VLSI Multiprocessor Workstation

M.D. Hill, S.J. Eggers, J.R. Larus, G.S. Taylor,
G. Adams, B.K. Bose, G.A. Gibson, P.M. Hansen, J. Keller,
S.I. Kong, C.G. Lee, D. Lee, J.M. Pendleton, S.A. Ritchie,
D.A. Wood, B.G. Zorn, P.N. Hilfinger, D.A. Hodges,
R.H. Katz, J. Ousterhout, and D.A. Patterson



Report No. UCB/CSD 86/273

December 1985

Computer Science Division (EECS)
University of California
Berkeley, California 94720

SPUR: A VLSI Multiprocessor Workstation [1]

M.D. Hill, S.J. Eggers, J.R. Larus, G.S. Taylor,
G. Adams, B.K. Bose, G.A. Gibson, P.M. Hansen, J. Keller, S.I. Kong,
C.G. Lee, D. Lee, J.M. Pendleton, S.A. Ritchie, D.A. Wood, B.G. Zorn,
P.N. Hilfinger, D.A. Hodges, R.H. Katz, J. Ousterhout, and D.A. Patterson

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

ABSTRACT

SPUR (Symbolic Processing Using RISCs) is a workstation for conducting parallel processing research. SPUR contains 6 to 12 high-performance homogeneous processors connected with a shared bus. The number of processors is large enough to permit parallel processing experiments, but small enough to allow packaging as a personal workstation. The restricted processor count also allows us to build powerful RISC processors, which include support for Lisp and IEEE floating-point, at reasonable cost. This paper presents a specification of SPUR and the results of some early architectural experiments. SPUR features include a large virtually-tagged cache, address translation without a translation buffer, LISP support with datatype tags but without microcode, multiple cache consistency in hardware, and an IEEE floating-point coprocessor without microcode.

KEY WORDS AND PHRASES: Address Translation, Cache, Cache Consistency, IEEE Floating-point, Lisp, Multiprocessor, RISC, Shared-Bus, Tagged Architecture.

1. Introduction

SPUR (Symbolic Processing Using RISCs) is a multiprocessor workstation being developed at U.C. Berkeley as a vehicle for conducting 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 [2]. This paper concentrates on the initial architectural research and development of SPUR. After the SPUR hardware and software are completed, additional papers will present the experimental results derived from using SPURs.

---

[1] SPUR is sponsored by DARPA under contract order 482427-25840 by NAVALEX. Additional computer resources provided by DARPA (order #4871) monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089.

[2] In this paper, we distinguish between *multiprocessing* and *parallel processing*.

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 [Demi82]. The complexity of managing such details has left parallel processing a novelty, rather than the norm. Consequently, SPUR is being designed 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 (e.g., message-passing).

The second observation is that hardware is more difficult to design and construct than is most software. Consequently, most SPUR hardware features are simple and frequently-used primitives. Features are migrated from software into hardware only if doing so achieves a significant performance gain for reasonable design and manufacturing costs. The complex hardware features included either facilitate parallel processing (e.g., hardware-based cache consistency) or make large contributions to performance (e.g., the instruction buffer and Lisp datatype tags).

The SPUR Processor is the third generation of RISC CPUs designed and implemented at U.C. Berkeley. It extends the work of RISC [Patt82] and SOAR [Unga84] with some special support for two emerging standards: Common Lisp [Stee84] and IEEE Standard 754-1985 for Binary Floating-Point Arithmetic [IEEE85]. Lisp and floating-point support has been designed 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.

The SPUR project is composed of SPUR workstation development work and research efforts in integrated circuits, computer architecture, operating systems, and programming languages. Integrated circuit researchers are examining CMOS design styles, the effects of scaling VLSI circuits, and control and clocking issues. Computer architecture researchers are studying multiprocessor address trace analysis, cache consistency, virtual 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.

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.

The rest of this paper presents a specification of SPUR and the results of some early architectural experiments. Sections 2 and 3 survey the system and processor architectures. Section 4 examines the memory system, discussing the memory model, the instruction buffer, the cache, in-cache address translation, and hardware support for cache consistency. Section 5 examines the CPU and floating-point coprocessor (FPU) architectures that support general-purpose computing, Lisp, and floating-point arithmetic. Section 6 gives the status of the project and conclusions.

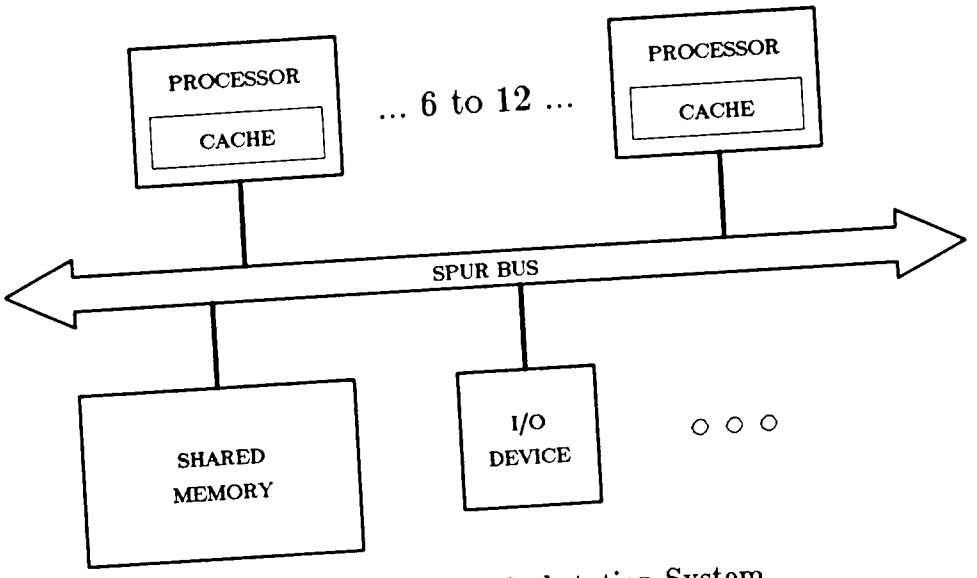


Figure 1. SPUR Workstation System

SPUR is a shared-bus multiprocessor. The system supports several identical high-performance processors on a modified Texas Instruments NuBUS [Texa83]. Each of the custom processors contains a large cache to reduce the bandwidth required from the bus and shared memory. Standard NuBUS I/O devices and gateways to other busses are also attached to the SPUR Bus to complete the system.

2. 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 restricted processor count also allows us to build powerful processors, which include support for Lisp and IEEE floating-point, at reasonable cost.

The processors are connected to each other, to standard memory, and to I/O devices with a modified commercial bus. Using a commercial bus will reduce prototype design time by allowing the use of standard sub-systems 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 and memory contention. 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 address 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 software without considering the existence of cache memory.

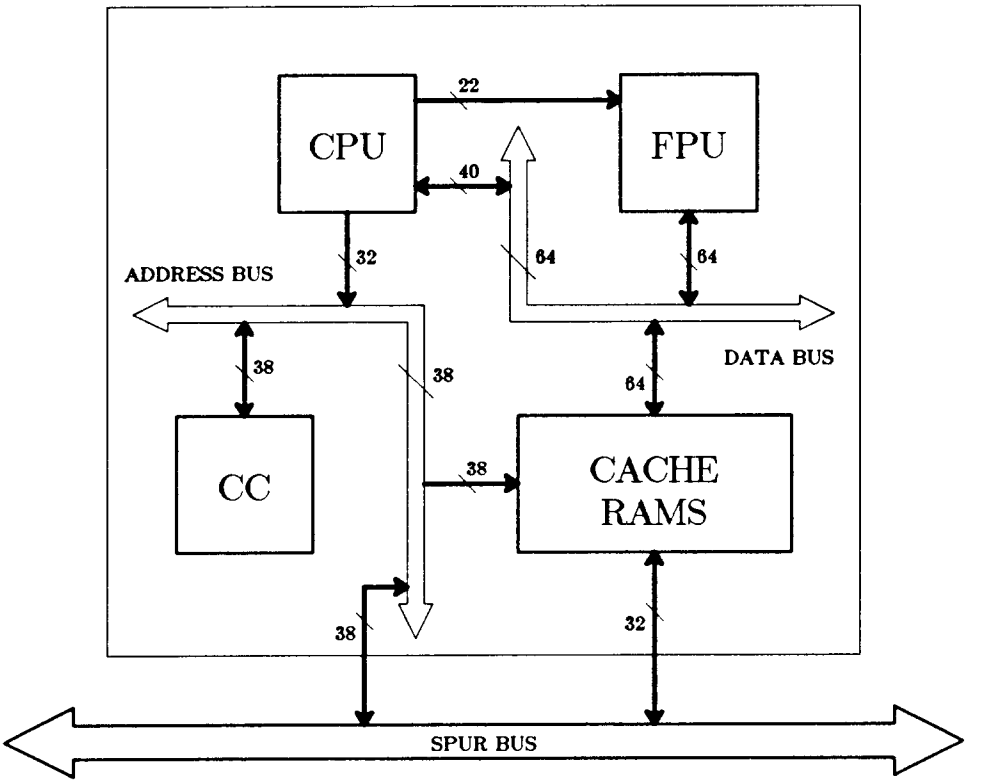


Figure 2. SPUR Processor Board

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 connect functional components together (not shown), and in CACHE RAMS to hold the state, address tags, and data of the cache. Memory addresses and data are handled on separate busses. 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 datapaths are not shown.

	Execution time (seconds)			Execution time ratio	
Gabriel Benchmark	DEC 8600	Symbolics 3600	SPUR (projected)	8600/ SPUR	3600/ SPUR
boyer	12.18	9.40	5.03	2.42	1.87
browse	38.69	21.43	-	-	-
ctak	2.32	5.04	1.55	1.50	3.25
dderiv	6.58	3.89	1.13	5.82	3.44
deriv	4.27	3.79	0.97	4.40	3.91
destru	2.10	2.18	0.46	4.57	4.74
div2	1.65	1.51	2.91	**0.57**	**0.52**
fft	9.08	3.87	9.38	**0.97**	**0.41**
file_print	1.08	2.60	-	-	-
file_read	2.34	4.60	-	-	-
frpoly(fixnum,p=15)	4.13	2.65	2.02	2.04	1.31
frpoly(bignum,p=10)	1.40	2.10	9.24	**0.15**	**0.23**
frpoly(flonum,p=15)	5.84	3.04	2.74	2.13	1.11
puzzle	15.53	11.04	7.11	2.18	1.55
stak	1.41	2.30	1.06	1.33	2.17
tak	0.45	0.43	0.12	3.75	3.58
takl	2.03	4.95	0.82	2.48	6.04
takr	0.81	0.43	0.24	3.38	1.79
terminal_print	0.70	4.89	-	-	-
traverse	46.77	41.71	25.75	1.82	1.62
triangle	99.73	116.99	92.78	1.07	1.26
geometric mean				1.82	1.69

Table 1. Gabriel Benchmark Results

This table presents Gabriel benchmarks [Gabr85] execution times for 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 [Gabr85]. The preliminary SPUR times are gathered with a functional-level simulator of a single processor, assuming a 150 nanosecond 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 in the table.

3. Processor Overview

A SPUR processor is implemented on a single board with about 200 standard chips and three custom 2-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 [Patt85,Patt82]. Like the RISC II implementation [Kate83a], the SPUR CPU uses a simple and 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. Common operations are executed by the FPU under hard-wired control. Infrequent operations cause traps and are handled by software.

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

4. The Memory System

The SPUR memory system appears to software as flat, global, shared memory (Section 4.1), but is implemented with a hierarchy of levels. The fastest level, the *instruction buffer*, is an instruction cache on the CPU chip (Section 4.2). The second level, the *cache*, is a cache on the processor board for instructions and data (Section 4.3). If information is not found in either of these local memories, then the virtual address is translated into a physical address (Section 4.4) 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 (Section 4.5).

4.1. The Memory Model

SPUR presents software with a 256-gigabyte global virtual address space, divided into 256 1-gigabyte segments. Every process has direct access to four segments via a 32-bit process-specific virtual address. 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, 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 independent 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.

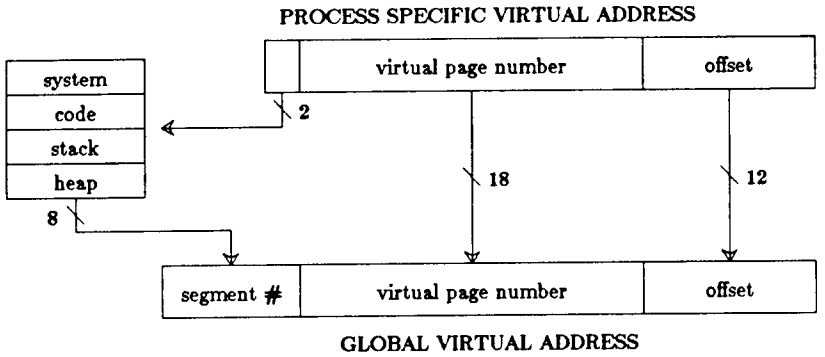


Figure 3. Virtual Memory Structure

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 (8 bits) in the global virtual address space. The other 30 bits are used directly for the displacement within the selected 1-gigabyte segment.

The global virtual address space limit of 256 segments was set by balancing the projected needs of software with the cost of hardware implementation. The segment limit does not constrain the number of *light-weight* processes, which use the address space of their parent. This is important since we expect the parallel processing Lisp system to make extensive use of light-weight processes. This limit does, however, restrict the number of concurrently active heavy-weight processes (e.g., 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.

4.2. The Instruction Buffer: An On-Chip Instruction Cache

The instruction buffer is a 512-byte instruction cache on the CPU chip. Its purpose is to reduce contention for the cache so that data references can use the single cache port without stalling the execution pipeline (see Section 5.1.2). By enabling instruction fetches and data references to be satisfied in parallel, the instruction buffer creates the illusion of a second cache port. In addition, the instruction buffer reduces effective instruction access time. This effect is not too important in SPUR because the cache can be accessed in approximately one cycle. Nevertheless, this effect will become come increasingly important as technological improvements reduce cycle times faster than inter-chip communication times.

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* [Good83, Hill84] (see Figure 4). Preliminary estimates show that the instruction buffer satisfies at least 75 percent of instruction fetches without cache accesses [Katz85a].

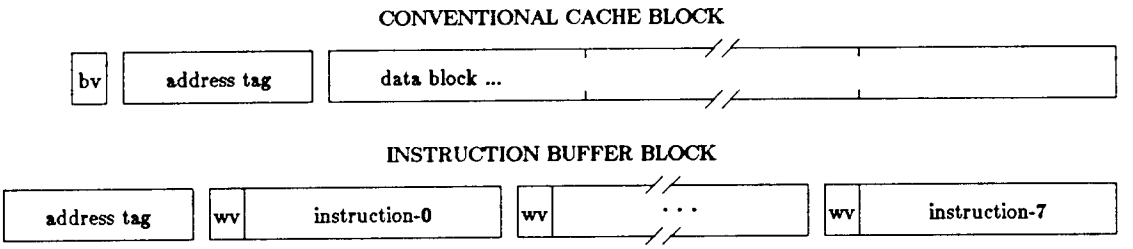


Figure 4. Cache Block with Sub-blocks

A conventional cache block (top) has three parts: the cache state bits (labeled *bv* for *block-valid* bit), the address tag, and the cache data block. The state bits record whether the information in the block is valid (or dirty if the cache is not instruction-only); 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, the 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 (address tag does not match) and sub-block misses (tag matches but 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 the feature [Hill84].

The instruction buffer miss ratio is reduced 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 begin 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.

4.3. The Cache

A 128K-byte cache for instructions and data is present on every processor board to reduce SPUR Bus traffic. For a fixed transfer size, bus traffic can be reduced by increasing a cache's size or complexity (e.g., associativity) [Smit82]. Memory chip technology makes it possible to build larger, unsophisticated caches

with fewer packages than smaller, more complex ones. Consequently, the SPUR cache is larger than the caches in most mainframes, but is simple. 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 the bus traffic. Simulation results find cache miss ratios under 2 percent (see column *Ideal* of Table 2) [Katz85a].

The cache in SPUR 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, many schemes that permit parallel address translation limit the size of the cache [3]. Consequently as cache sizes increase, we believe virtually-tagged caches will have shorter accesses times than physically-tagged caches. In addition, virtually-tagged caches permit address translation to be done more slowly than a cache access since translation need not be done on every cache access. SPUR exploits this freedom by eliminating the traditional translation buffer (Section 4.4).

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 virtual-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.

The first problem is handling the virtual address space changes associated with most context switches. A virtual address space change means that virtual addresses now refer to new locations. A virtual-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 attaching address space identifiers to cached data or by flushing old data on every context switch. The former method increases cache complexity, and the latter reduces performance for large caches. SPUR avoids the problems of virtual address space changes by using a global virtual address space that does not change after a context switch.

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 one off the copies is modified. To preserve the programmer's model of memory, the virtually-tagged cache must guarantee that a

[3] At present, many machines overlap address translation with physical cache lookup by starting the cache lookup with address bits within the page offset, because these bits do not change in address translation. This only works if the cache size divided by its associativity is equal to or smaller than the page size. As cache sizes increase, systems will have to do address translation before the cache lookup unless they constrain the mapping of virtual pages to physical page frames or use virtual address tags.

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 (i.e. the same global virtual address). Software resolves the location of static, shared information at load-time, and uses operating systems calls that allocate new storage to establish the location of dynamic, shared information.

The third problem with virtually-tagged caches is updating data in the cache 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 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 (Section 4.5). 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. The latter does not imply a complete cache flush since the cache supports flushing of individual blocks.

4.4. Address Translation without a Translation Buffer

The mapping of virtual addresses to physical addresses is usually maintained in a structure called a *page table*. 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 [Wood86]. The performance of SPUR *in-cache* translation is, at least, comparable to that with fixed-size translation buffers, and in-cache translation has two advantages. First, it saves the design and implementation costs of a translation buffer. Second, it keeps PTEs consistent (translation buffer consistency) by storing PTEs in a cache that keeps data consistent.

When data is referenced that is not in the cache, address translation is done by the cache controller as follows [4]. 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.

---

[4] In this discussion, *data* refers to instructions and data, in contrast to address

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 non-pageable 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. The increase was 0.00142 (from 0.01677 to 0.01819).

4.5. Cache Consistency Hardware

The difficulty of maintaining the shared memory model in multiprocessor systems that 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, which is not feasible in a system with several high-performance processors [Cens78]. The second approach requires software to identify that 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 [Stan85]. The principle weakness of the software approach is that it may require extra effort from the programmer, thereby potentially 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 [Katz85b]. The responsibility for maintaining consistency is distributed among the caches. If a cache owns a block, then there are no copies of the block 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

translation information, i.e., PTEs.

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	IBM 3033 TLB
LISZT	0.00584	0.00610 (1.000)	0.00775 (1.270)	0.00614 (1.006)
VAXIMA	0.01844	0.01875 (1.000)	0.02432 (1.297)	0.02001 (1.067)
MVS	0.01677	0.01981 (1.000)	0.02208 (1.115)	**0.01769 (0.893)**

Table 2. In-Cache Address Translation vs. 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 misses of 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 is done without cost. The second alternative, *SPUR in-cache*, uses the cache to hold PTEs for 4K-byte pages. The third and fourth alternatives use translation buffers to do translation. The third uses half of the VAX-11/780 Translation Buffer (128 entries, 512-byte pages, and 2-way set-associative) because the VAX-11 restricts process and system entries to different halves of the buffer. The fourth uses the IBM 3033 Translation Lookaside Buffer (128 entries, 4K-byte pages, 2-way set-associative). In all but one case (in bold), SPUR in-cache translation performs slightly better than systems that included expensive translation buffer hardware.

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 is used for broadcasting ownership requests and transferring cache blocks. Most bus transactions begin with a type field (e.g., read, read-for-ownership) and a block address (both virtual and physical), and end with a data transfer [Gibs85]. 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 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 and flexible mechanism for sharing data between processes. Second, it is implemented in hardware. This 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 [5]. 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 inter-processor sharing can occur, no consistency-preserving bus transactions will be made. Fifth, the algorithm is not too 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.

5. The CPU and Floating-Point Coprocessor

The SPUR CPU design evolved from the RISC II design [Kate83a]. Like RISC II, SPUR has a large register file with multiple, overlapping register windows to speed up procedure calls and a streamlined instruction set. 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 that are implemented in a short, uniform pipeline. Every instruction uses a particular resource in the same pipeline stage. For example, all SPUR instructions use the ALU to combine operands or calculate an effective address in the second stage of

[5] 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.

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. 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.

This section describes the CPU and FPU of the SPUR in three parts: the general-purpose features, architectural support for Lisp, and the floating-point implementation.

5.1. General-Purpose Features

5.1.1. The Register State

The SPUR register state, shown in Figure 6, includes 32 general-purpose registers. Like the RISC II chip, the SPUR CPU contains several copies of the general-purpose register set (not shown in Figure 6) 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 cost of the multiple register sets is primarily a significant amount of chip area and, to a lesser extent, slower register access time and increased process switching overhead.

5.1.2. The Execution Pipeline

The SPUR execution pipeline is one stage longer than the three-stage RISC II pipeline (see Figure 7). RISC II could issue a register-to-register instruction every cycle. Resources were used 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 a four-stage pipeline to attempt to issue and complete an instruction every cycle. The new pipeline stage allows memory referencing instructions to make cache accesses and forces register-to-register instructions to delay their register write for one stage. Thus, 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

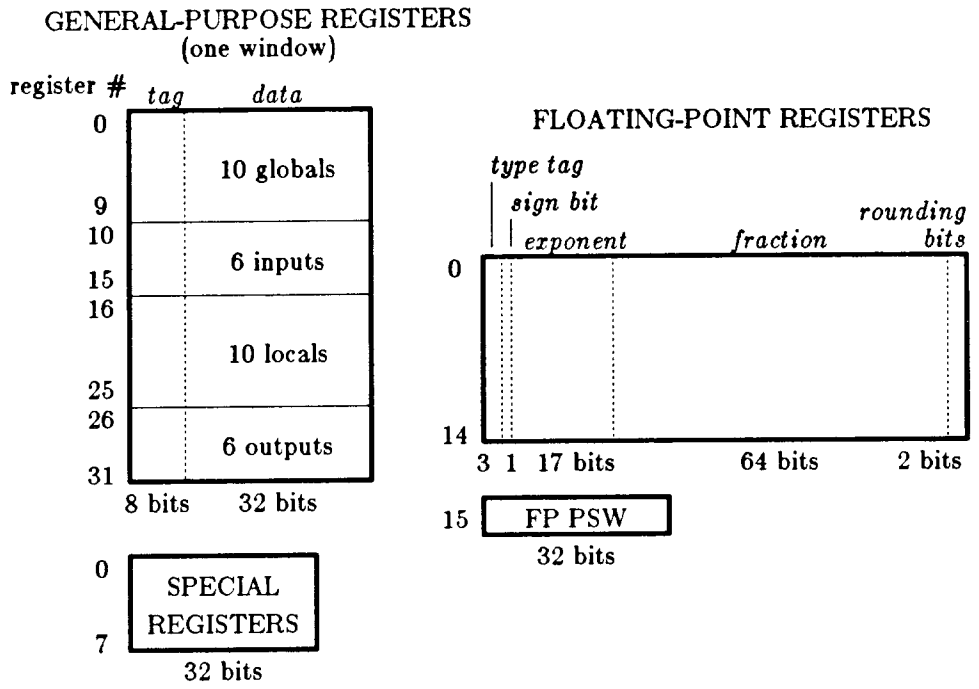


Figure 6. SPUR Registers

SPUR's registers are divided into three groups: general-purpose, special, and floating-point. The general-purpose registers are organized into fixed-sized 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 eight-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 3-bit type tag to simplify detection of infrequent floating-point types (e.g., 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 was not implemented, because of insufficient FPU chip area.

cycles with instruction buffer and cache miss ratios of 14 and 1 percent. 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.

5.1.3. The Instruction Set

This section focuses on a few important decisions in the SPUR instruction set. See [Tayl85] for the complete design. All instructions are four bytes long, and use fixed positions for the opcode and register specifiers to simplify decoding.

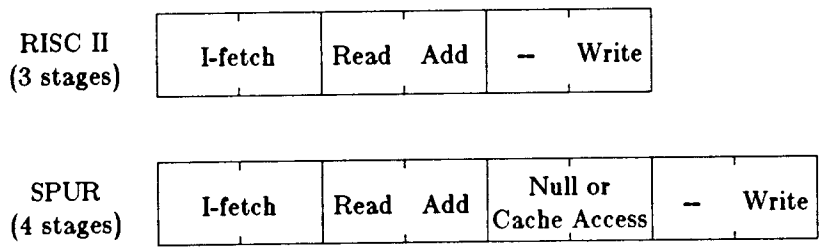


Figure 7. RISC II and SPUR Pipelines

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; and 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.

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.

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 a register and an immediate displacement. SPUR uses a *delayed* load that requires software to not use the incoming data in the next instruction executed. Cache misses on data reference 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. Initially stores did not stall the pipeline, because the cycle time was set to the cache write time. We were able to improve performance by reducing the cycle 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.

Instruction	Operands	Action	Cycles
LOAD/STORE			
load_32	dest, src1, ri	dest ← M[src1 + ri]	1
load_external	dest, src1, ri	dest ← external 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 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 else change next instruction into no-op	1
jump	address	pc ← word address (in same segemnt)	1
jump_register	src1, ri	pc ← src1 + ri	1
CALL/RETURN			
call, call_kernel	address	increment current window pointer; save pc; pc ← word address	1
return, return_trap, return_kernel	src1, ri	pc ← src1 + ri 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 3. Basic SPUR Instructions

This table lists the basic SPUR instruction set. The column *cycles* is 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 5-bit register specificers. *Ri* is either a 5-bit register specifier or a 14-bit signed immediate constant. *Rci* stands for a 5-bit register specifier or a 5-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 [Tayl85].

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 [Gros83]. 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 [LLNL84]. This variant of the delayed branch makes it 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 [Kate83b]. 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.

5.2. Supporting Lisp

The Lisp programming language has some features that are 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 (e.g., Symbolics 3600 [Moon85] , Lambda [LMI83] , and the Xerox D-Machines [Burt80]). Our approach emphasizes a simple, regular instruction set, overlapping register windows, and tagged data. Table 4 lists the instructions tailored for Lisp.

Fast function calls and returns are particularly important for Lisp, because Lisp programs are constructed out of many, small functions. SPUR provides fast function calls and returns through the overlapping register window mechanism. Studies have shown that this mechanism, which was developed for C, is effective at speeding up Lisp calls and returns [Pond83]. The complicated argument options allowed by Common Lisp (e.g., 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.

5.2.1. 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 datatype. 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 datatype tags of operands in parallel with operating on the 32-bit data values (see Figure 8). Type checking in SPUR assumes that most arithmetic operands are integers.

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) pc ← pc + signed word offset	1
tag_cmp_branch_likely	cond, src1, tag_imm, offset	if (src1<tag> cond tag_imm) pc ← pc + signed word offset 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

Table 4. SPUR Lisp Instructions

This table lists Lisp instructions. The column *cycles* is 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.

Similar approaches have been used on several other machines [Road83, Unga84]. For example, a polymorphic add operation is implemented with an add instruction that begins by adding the 32-bit operands as if they were integers and, in parallel, checking the datatype tags to verify that they are integers. If both operands are integers, then 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 datatype 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 that are not handled by the hardware.

Datatype tags also assist list manipulation, which is fundamental in Lisp. A list is a sequence of elements (e.g., *(a b c)*). The Lisp functions that manipulate lists are called *CAR* and *CDR*. *CAR* returns the first element of a list (e.g., *a*), and *CDR* returns the rest of the list (e.g., *(b c)*). CAR and CDR can be implemented with load instructions since lists are stored as linked-lists in main memory. However, the semantics of Common Lisp strongly encourage that an exception be generated 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 *CARs* and *CDRs* in a correct program are lists. SPUR provides a *cxr* (car or cdr) instruction, which checks the

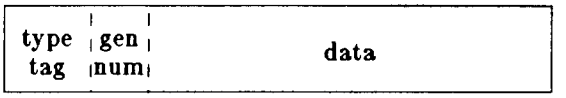


Figure 8. Lisp Tagged Data

SPUR augments Lisp data words with an eight-bit tag that includes a six-bit datatype 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.

datatype tag in parallel with the load. A trap is generated if the type of either 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 be able to 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 2-bit *generation number* in the tag to assist a *generation scavenging* garbage collection algorithm [Unga84]. 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.*

5.2.2. Poor Data Density

The SPUR architecture has been designed with more emphasis on speed and simplicity than concern for code or data density. The prototype implementation has particularly poor Lisp data density because of a decision not to build a complete 40-bit system.

The CPU manipulates 40-bit data (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. There are three approaches 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. A 40-bit-word memory system was rejected, because it would preclude the use of many off-the-shelf sub-systems, which would substantially delay completion of the hardware. It would also have complicated non-Lisp software in such areas as string manipulation and file transfer with non-SPUR machines. Permitting unaligned cache accesses was rejected because of the

complexity it would add to the cache. An unaligned access can cross a cache block boundary, possibly forcing the cache to handle two cache misses, including 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, or some Lisp data within 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 prototype.

5.3. Floating-Point Support

SPUR implements the IEEE 754 binary floating-point standard [IEEE85] 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 the first implementation of IEEE floating-point that does not use any microcoded control.

5.3.1. 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, divide, and two types of compares that are similar to their integer counterparts. Except for multiply (7 cycles) and divide (19 cycles), a new floating-point instruction can be issued every four cycles.

Data is 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.

The FPU contains fifteen 87-bit floating-point registers organized as a single register set (see Figure 6). 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: to reduce access time for floating-point operands, to allow more freedom in setting the width of floating-point registers, to permit concurrent execution of integer and floating-point operations, and 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 part 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.

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, src1, 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
fp_cmp_branch_delayed	cond, src1, src2, offset	if (FPU src1 cond FPU src2) pc ← pc + signed word offset	4
fp_cmp_branch_likely	cond, src1, src2, offset	if (FPU src1 cond FPU src2) pc ← pc + signed word offset else change next instruction into no-op	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, src1	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

Table 5. SPUR Floating-Point Instructions

This table lists SPUR floating-point instructions. The column *cycles* is the minimum number of cycles consumed by an instruction in normal operating mode. If the FPU and CPU are operated concurrently, then a CPU instruction can begin one cycle after an FPU instruction has started (see Section 5.3.2). 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. The *to_cpu* and *from_cpu* instructions transfer integers directly between the integer and floating-point register sets so that the FPU can be effectively used for integer multiply and divide. Most floating-point operations execute in four cycles using the add/subtract hardware. Multiply and divide use additional special-purpose hardware. The *sync* instruction is used when the CPU and FPU are executing instructions in parallel and the CPU must wait until the FPU is not busy.

The FPU manipulates single (32 bits), 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 3-bit tag is included to identify 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 three bits 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. 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.

5.3.2. 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 [Inte85]. 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 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 [Cass84]. 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 datapath between the FPU and the memory system is 64 bits wide. This allows load and store instructions to move single and double-precision numbers with a single transfer and extended-precision numbers with two transfers. Commercial FPU interfaces have only recently become 32 bits wide [Cass84]. 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 [Hans85]. 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 in memory if the store could issue before the operation completes.

6. Status and Conclusions

The implementation of SPUR is in progress. As of November 1985 most of the custom components have been described at the register-transfer level with a variant of the ISP language and simulated with a software package called N.2 [Ordy83]. The processor board has been designed, simulated with N.2, but not yet implemented. The layouts of the CPU and FPU chips are near completion. Both use four-phase non-overlapping clocks with goal of a 150 nanosecond cycle time. All datapaths and much of the control have been implemented. The CC chip control has been specified, but little of the simple datapath has been implemented. We expect to have working components by late-1986, and a working system in 1987.

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 keep 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, datatype 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 datapath.

The goal of the first phase of the SPUR Project is to design and implement one or two working prototypes. We hope to complete this goal by the end of 1987. If the prototypes meet our expectations, we hope to find partners to help us transfer SPUR from academia to industry.

7. Acknowledgements

The SPUR project is a cooperative project that benefits from contribution 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, Gregg Foster, Jim Goodman, Robert Henry, Louis Monier, Prabhakar Ragde, Jim Smith, and Chuck Thacker for their suggestions that improved the quality of this paper.

SPUR was first presented at the 1985 Asilomar Microcomputer Workshop by the first four authors of this paper. SPUR is sponsored by DARPA under contract order 482427-25840 by NAVALEX.

References

Burt80.
R.R. Burton, R.M. Kaplan, L.M. Masinter, B.A. Sheil, A. Bell, D.G. Bobrow, L.P. Deutsch, and W.S. Haugeland, *Papers on Interlisp-D*, September 1980. Xerox PARC Technical Report SSL-80-4

Cass84.
Barbara A. Cassel, *MC68020 32-Bit Microprocessor User's Manual*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

Cens78.
Lucien M. Censier and Paul Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Transactions on Computers*, vol. C-27, no. 12, pp. 1112-1118, December 1978.

Demi82.
J. Deminet, "Experience with Multiprocessor Algorithms," *IEEE Trans. on Computers*, vol. C-31, no. 4, April 1982.

Gabr85.
R.P. Gabriel, *Performance and Evaluation of Lisp Systems*, MIT Press, 1985.

Gibs85.
G. Gibson, "SPURBUS Specification," *Proc. of CS292i: Implementation of VLSI Systems*, R.H. Katz, Ed., University of California, Berkeley, September 1985. Computer Science Division Technical Report UCB/CSD 86/259

Good83.
J.R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," *Proc. Tenth International Symposium on Computer Architecture*, pp. 124-

131, Stockholm, Sweden, June 1983.

Gros83.
T. Gross, *Code Optimization of Pipeline Constraints*, August 1983. Ph.D. Thesis, Stanford University

Hans85.
P.M. Hansen, *Coprocessor Architectures for VLSI*, University of California, Berkeley, May 1985. Unpublished Thesis Research Proposal

Hill84.
M.D. Hill and A.J. Smith, "Experimental Evaluation of On-Chip Microprocessor Cache Memories," *Proc. Eleventh International Symposium on Computer Architecture*, Ann Arbor, MI, June 1984.

IEEE85.
IEEE, *IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*, 1985. Order number CN953

Inte85.
Intel, in *Microsystem Components Handbook, Volume I*, pp. 3.175-3.197, Santa Clara, CA, 1985. 8087 Numeric Data Coprocessor

Kate83a.
M.G.H. Katevenis, R.W. Sherburne, D.A. Patterson, and C.H. Séquin, "The RISC II Micro-Architecture," *Proc. VLSI 83 Conference*, Trondheim, Norway, August 1983.

Kate83b.
M.G.H. Katevenis, *Reduced Instruction Set Computer Architectures for VLSI*, October 1983. Ph.D. Thesis, U.C. Berkeley

Katz85a.
R.H. Katz, S.J. Eggers, G.A. Gibson, P.M. Hansen, M.D. Hill, J.M. Pendleton, S.A. Ritchie, G.S. Taylor, D.A. Wood, and D.A. Patterson, *Memory Hierarchy Aspects of a Multiprocessor RISC: Cache and Bus Analyses*, University of California, Berkeley, January 1985. Computer Science Division Technical Report UCB/CSD 85/221

Katz85b.
R.H. Katz, S.J. Eggers, D.A. Wood, C.L. Perkins, and R.G. Sheldon, "Implementing a Cache Consistency Protocol," *Proc. Twelfth International Symposium on Computer Architecture*, Boston, June 1985.

LLNL84.
LLNL, *S-1 AAP Review*, November 1984. Physics Briefing 85-606

LMI83.
LMI, *The Lambda System: Technical Summary*, 1983. LISP Machines, Inc.

Moon85.
D.A. Moon, "Architecture of the Symbolics 3600," *Proc. Twelfth Symposium on Computer Architecture*, Boston, MA, June 1985.

Ordy83.
G.M. Ordy and C.W. Rose, "The N.2 System," *Proc. of 20th ACM/IEEE Design Automation Conference*, pp. 520-526, 1983.

Patt82.
D.A. Patterson and C.H. Séquin, "A VLSI RISC," *Computer*, vol. 15, no. 9, pp. 8-21, September 1982.

Patt85.
D.A. Patterson, "Reduced Instruction Set Computers," *Comm. ACM*, pp. 8-21, January 1985.

Pond83.
C. Ponder, *But will RISC run LISP? (a feasibility study)*, University of California, Berkeley, April 1983. Unpublished Masters Report

Road83.
C.B. Roads, *3600 Technical Summary*, Cambridge, MA, 1983. Symbolics, Inc.

Smit82.
A.J. Smith, "Cache Memories," *Computing Surveys*, vol. 14, no. 3, pp. 473 - 530, September, 1982.

Stan85.
Stanford, *Biannual Research Summary, DARPA VLSI Contracts*, November 1984 - March 1985. Stanford CSL & ICL

Stee84.
G.L. Steele, *Common LISP: The Language*, Digital Press, Burlington, MA 01803, 1984.

Tayl85.
G. Taylor, "SPUR Instruction Set Architecture," *Proc. of CS292i: Implementation of VLSI Systems*, R.H. Katz, Ed., University of California, Berkeley, September 1985. Computer Science Division Technical Report UCB/CSD 86/259

Texa83.
Texas-Instruments, *NuBUS Specification*, 1983. TI-2242825-0001

Unga84.
D. Ungar, R. Blau, P. Foley, D. Samples, and D. Patterson, "Architecture of SOAR: Smalltalk on a RISC," *Proc. Eleventh International Symposium on Computer Architecture*, pp. 188-197, June 1984.

Wood86.
D.A. Wood, S.J. Eggers, G. Gibson, M.D. Hill, J. Pendleton, S.A. Ritchie, R.H. Katz, and D.A. Patterson, "An In-Cache Address Translation Mechanism," *Submitted to Thirteenth International Symposium on Computer Architecture*, Tokyo, Japan, June 1986.