# SPUR Memory System Architecture

*David A. Wood*
*Susan J. Eggers*
*Garth Gibson*

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

Version 2.1: 1/7/88

## ABSTRACT

This document describes the memory system architecture of the SPUR workstation. SPUR is a bus-based multiprocessor, with caches to reduce each processor's bandwidth requirement. A hardware cache coherency protocol maintains a consistent image of memory across all the caches. A novel address translation scheme eliminates the need for translation buffers.

This document is intended as a reference for system and diagnostic programmers. It describes the cache coherency protocol, address translation algorithm, and exception handling mechanisms in detail.

## Acknowledgements

Many people have contributed to the design of SPUR. It is not possible to single out the contributions of every member of the project over its three year history. We would especially like to thank Deog-Kyoon Jeong and Walter Beach for their work on the cache controller chip implementation, and Ken Lutz for his work on the processor board. We would also like to thank Randy Katz, Corinna Lee, and Doug Johnson for their helpful comments on earlier drafts of this report.

This research was funded by DARPA contract number N00039-85-C-0269 as part of the SPUR research project. Additional funding came from the IBM Pre-Doctoral Fellowship program, and the California MICRO program (in conjunction with Texas Instruments, Xerox, Honeywell, and Phillips/Signetics).

# Table of Contents

## List of Tables

## List of Figures

## 1. Introduction

This document describes the memory system architecture of the SPUR workstation. SPUR is a multiprocessor workstation based around a common bus and shared main memory [Hill86]. Caches are used to reduce each processor's bandwidth demand on the bus, and to reduce the average memory access time. A distributed cache coherency protocol, called *Berkeley Ownership* [Katz85], maintains consistency across all the caches in the system. A novel address translation scheme, called *In-Cache Address Translation* [Wood86], provides a single shared virtual address space to all processors.

Each SPUR processor contains 3 functional modules, a central processing unit (CPU), a floating point coprocessor (FPU), and a Cache Controller (CC). In the SPUR prototype, each of these is implemented as a single CMOS VLSI chip. Each processor is a single board, containing the custom chips, high-speed static RAMs, and bus interface logic.

This document is intended as a reference for system and diagnostic programmers. The cache coherency protocol, address translation algorithm, and exception handling mechanisms are all described in detail. The rationale behind the important design decisions is presented.

### 1.1. Virtual Address Cache

An unusual feature of SPUR's memory architecture is its virtual address cache. The cache associates virtual address tags, rather than physical address tags, with blocks of data. This allows cache hits to proceed without address translation. In contrast, physical address caches require that address translation be done before or in parallel with the tag lookup. For this reason the cycle time can be shorter with a virtual address cache, improving system performance.

In addition to the cycle time advantage, virtual address caches allow address translation to be done more slowly, since translation need not be done on every cache access. SPUR exploits this freedom by eliminating the traditional translation buffer and uses *In-Cache Address Translation* instead (see Section 2).

Virtual address caches are not commonly used because they can suffer from the so called synonym problem. A *virtual address synonym* exists when two virtual addresses map to the same physical address. If the virtual address is used to index the cache, the same datum could reside in two (or more) different cache blocks for certain cache configurations. Inconsistencies can arise if one copy is modified and the other is not. SPUR avoids these problems by disallowing synonyms (enforced by the operating system). SPUR provides a *global virtual address space*, shared by all processes. Thus, two processes must access a shared datum using the same global virtual address.

### 1.2. Virtual Memory

The mapping from virtual to physical addresses is maintained in a structure called a *page table*. This table resides at a well-known location in the virtual address space; during address translation, we locate the appropriate page table entry (PTE) by indexing into the table using the virtual page number (the high-order address bits). Computers with physical address caches use a special-purpose cache for PTEs, called the *translation look-aside buffer* (TLB), to reduce address translation time. Because of SPUR's virtual address cache, address translation is necessary only on cache misses. Rather than having a separate TLB, the cache acts as both a TLB and a data cache. The details of the algorithm are explained in Section 2.

Simulation studies indicate that in-cache translation performs (at least) as well as TLBs, while eliminating the extra hardware. Additionally, it eliminates the multiprocessor TLB consistency problem. Since TLBs are merely special-purpose caches, multiprocessors that use a TLB per processor suffer from a TLB consistency problem (analogous to the data cache consistency problem discussed below). The SPUR in-cache translation mechanism avoids this problem by storing the PTEs only in the cache, where they are kept consistent by the regular coherency protocol.

### 1.3. Cache Coherency

A multiprocessor system with caches must maintain *cache coherency*, i.e., all processors must have a consistent view of memory. The SPUR workstation utilizes the *Berkeley Ownership* protocol [Katz85] to guarantee that every processor sees only the most recently written data. The protocol is distributed, meaning that all processors monitor the backplane and check their caches on each bus operation. It attempts to minimize bus traffic by supporting ''copy-back'' caches and cache-to-cache transfers on shared reads to dirty data.

The protocol's underlying principle is that a cache must explicitly *own* a memory block before it is permitted to update the block. Within a cache, a given block can exist in one of four states: Invalid (unaccessible), Unowned (readable but not writable), OwnShared (writable, but must communicate writes to other caches), or OwnPrivate (writable with no communication necessary). The protocol guarantees that a memory block has only one owner at a time, although multiple caches may have Unowned copies. Ownership carries with it the privilege to update a memory block, but also the responsibilities of providing the memory block to other processors on a read and eventually writing the block back to shared memory. The coherency protocol is described in more detail in Section 6.

### 1.4. SPURBus

A SPUR workstation consists of a set of processor boards connected by the *SpurBus* to memory and peripherals. The SpurBus is based on the Texas Instrument NuBus™ [Texa83], with extensions to support cache coherency. NuBus memory and peripherals are compatible with SpurBus, allowing us to obtain these components from Texas Instruments.

The NuBus is a simple, 32-bit wide synchronous bus. Running at 10Mhz, with 32 byte block transfers, and 300ns latency to the first word, its maximum bandwidth is 29MB/s. The pseudo-roundrobin arbitration policy provides (reasonably) fair access to all processors. Memory mapped I/O devices and interrupts simplify writing device drivers.

To support cache coherency, the SpurBus adds a separate *virtual bus* that runs semi-independently from the *physical* NuBus. The virtual bus is used only by the processor boards, and does not connect to memory or peripherals. On (most) bus operations, both the virtual and physical addresses are sent out on their respective busses. The memory (or peripheral device) always responds to the physical bus request; another processor *may* respond on the virtual bus, if required by the coherency protocol.

Accessing memory and devices over the SpurBus is described in Section 3.5. Details of the bus, including the protocol and arbitration, are described in the *SpurBus Specification* [Gibs85, draft specification].

## 1.5. Memory Access Instructions

SPUR is a load/store machine, along the lines of RISC-II [Kate83], and supports a myriad of load instructions [Tayl88]. The vanilla **ld_32** instruction loads a single, aligned 32-bit word from memory into a register. **ld_40** loads a 32-bit word, plus an 8-bit tag into a register; the data must be aligned on a 64-bit boundary. **cxr** is a variant of **ld_40** that performs an appropriate tag check. For the floating point coprocessor, there are four more load instructions, **ld_sgl**, **ld_dbl**, **ld_ext1**, and **ld_ext2**, that load data into the floating point registers.

These instructions each have an unusual variant. **ld_32_ro**, **ld_40_ro**, **cxr_ro**, **ld_sgl_ro**, **ld_dbl_ro**, **ld_ext1_ro**, and **ld_ext2_ro** perform their normal functions, but provide a hint to the memory system that the program may write the same word, actually the same cache block, in the near future. This information is used to improve the performance of the cache coherency protocol (see Section 6).

**ld_32_ri** is a privileged instruction that loads a 32-bit word, and *ignores page faults* in the process. This strange and dangerous instruction is required for certain virtual memory operations (see Section 7). Improper use of this instruction can obviously result in serious software errors.

The final load instruction, **ld_external**, in conjunction with **st_external**, provides access to *control space*, which is described in Section 4.

The store instructions are less numerous and less exotic than the load instructions. **st_32** and **st_40** store a 32-bit word and, in the later case, an 8-bit tag from a register to memory. The floating point store instructions, **st_sgl**, **st_dbl**, **st_ext1**, and **st_ext2**, store from the floating point registers to memory.

The **test_and_set** instruction provides a simple synchronization primitive. A word is read from memory and loaded into a register. At the same time, the memory word is set to 1. These two actions are guaranteed to be atomic (see Section 5.3.2 for exceptions), so spin locks can be implemented easily.

## 1.6. Address Spaces

The memory system supports three address spaces: virtual, physical, and control. All load and store instructions, except **ld_external** and **st_external**, can be used to access the virtual address space. Two bits in the kernel processor status word register (KPSW), control access between virtual and physical spaces. K_VirtIFetch indicates whether instruction fetches and prefetches should go to the virtual address space. K_VirtDFetch indicates whether **ld_32** and **st_32** access the virtual or physical address space. All other load and store instructions always go to the virtual address space. Control space is accessed exclusively by the **ld_external** and **st_external** instructions. Protection is only checked on virtual memory accesses; control space can only be accessed in kernel mode, and physical space should only be made accessible in kernel mode.

All user mode accesses, and most kernel mode accesses, will be to the virtual address space. The virtual address space is divided into segments, to allow inter-process sharing, and pages, to support demand paging of memory. Pages may be made non-cacheable, which is useful when accessing I/O device registers. Only the **ld_32** and **st_32** instructions are allowed to access non-cacheable pages, all other instructions will cause faults.

The physical address space provides direct access to main memory and I/O devices. In addition, most of the processor board's state is accessible this way. The physical address space is primarily intended for diagnostics, bootstrap code, and system error handling. The physical

address space is described in Section 3.

The third space, control space, provides access to control registers and cache management functions. The **ld_external** and **st_external** instructions are used solely for this purpose. Bits 4-2 of the effective address are interpreted as a *subcacheop*, specifying the request. Control space is discussed in detail in Section 4.

## 2. Virtual Memory Support

SPUR provides support for demand paged virtual memory. The cache controller performs address translation using the In-Cache Address Translation algorithm. Maintaining translation coherency is simplified because in-cache translation does not use a separate translation buffer. However, the virtual address cache is NOT transparent to system software, and therefore the operating system must enforce certain restrictions.

SPUR supports a single, segmented virtual address space, referred to as the *global virtual address space*, that is shared by all processes. It consists of 256 segments, each one gigabyte, for a total of 256 gigabytes of virtual storage. A processor has 32 bits of address, called the *process virtual address* (PVA), so it can only access four segments at one time. The top two bits of the process virtual address select one of four *active segments*; the operating system maintains the active segment mapping. By software convention, the 4 segments usually contain system code and data, user code, private data, and shared data, respectively. The mapping from process virtual address to global virtual address is described below.

Processes can share data at the segment level; if any portion of a segment is shared by two processes, the whole segment is shared. This is the only mechanism for sharing data, since synonyms are prohibited because of the virtual address cache. The operating system must not allow two global virtual addresses to map to the same physical address (i.e., a virtual address synonym) or inconsistencies may arise (see Section 7 for more details).

Segments are further divided into 4 Kbyte pages. Page attributes, such as physical address and protection, are maintained in a single pagetable, that maps the entire global virtual space. Each pagetable entry (PTE) is 4 bytes (see Section 2.3). Because the virtual space is 256 gigabytes, the pagetable requires 256 megabytes. Obviously the pagetable must reside in virtual space to reduce the physical memory requirements.

Because the pagetable is in virtual space, and it maps the entire virtual space, a portion of the pagetable maps the pagetable itself. This self-referential portion of the pagetable is called the *second-level* or *root* pagetable, and is illustrated in Figure 2.1. The root pagetable takes up 256 kilobytes, and contains the pagetable entries for each page in the pagetable.

Similarly, because the root pagetable is just a subset of the full pagetable, it is also mapped by a portion of itself. This *root*$^2$ pagetable (pronounced *root squared*) is only 256 bytes, and contains PTEs for the 64 pages of the root pagetable. The root$^2$ pagetable fits into a single page, and is mapped in turn by a single PTE, called the *root*$^3$ pagetable. The root$^3$ pagetable is the only PTE that maps the page in which it resides.

Figure 2.1 illustrates the mapping. At the left of the figure is the global virtual space. In this example, the pagetable resides at address 0, and occupies the lowest quarter of segment 0. The pagetable is shown expanded to the right of the global virtual space, as shown by the dashed lines. The pagetable can be thought of as containing 256 segment pagetables, that each map one of the global segments; the dotted lines illustrate that the segment i pagetable contains the mapping for segment i. The root pagetable is just a subset of the pagetable, and is shown expanded to the right of the pagetable. Again, the root pagetable can be thought of as containing 256 segment root pagetables, that contain the second level mappings for each of the segments. At the right of Figure 2.1, the root$^2$ pagetable is expanded. Each entry in this table maps a page of the root pagetable; or, in other words, it is the 3rd level mapping for 4 segments of data. One of these entries, in this example the entry at address 0, is the root$^3$ pagetable. This entry maps the

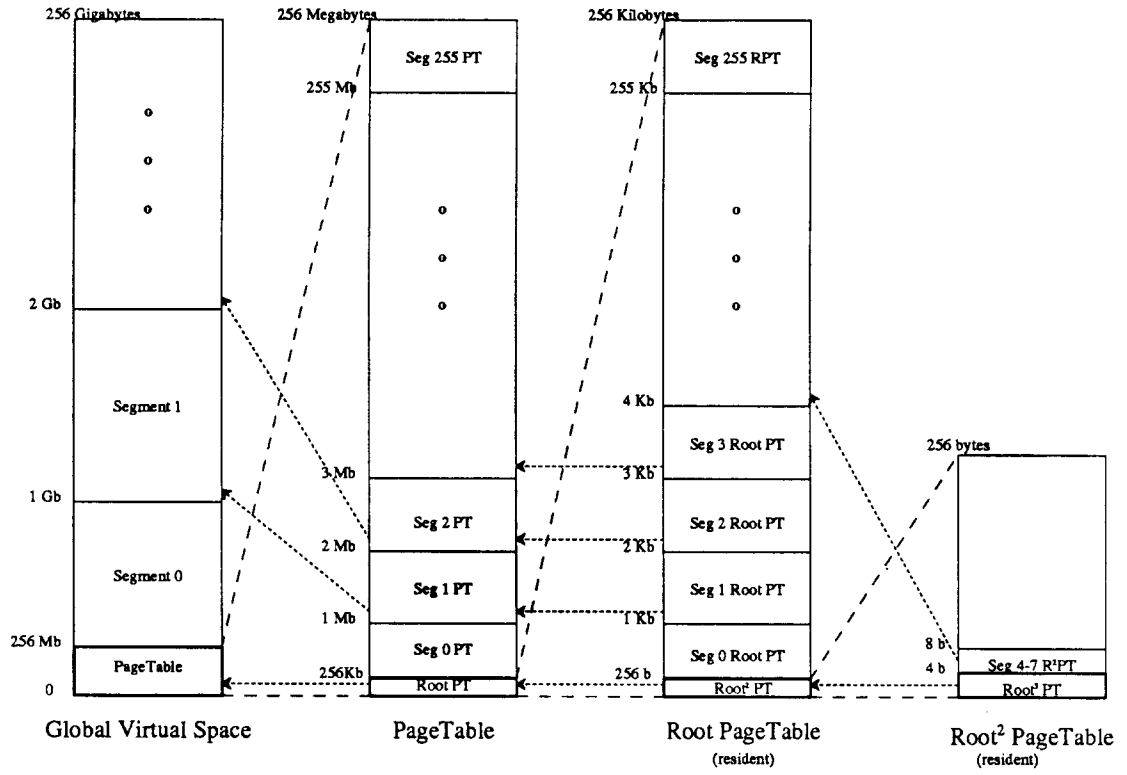


Figure 2.1: Self-referential Mapping of Pagetables

segment that contains the pagetable (plus the three adjacent segments).

In a traditional multi-level pagetable scheme, the translation hardware would start with the root[3] pagetable and work down to the 1st level of the pagetable. However, because the pagetable is contiguous at a known location in virtual space, it is possible to go from a virtual address directly to the virtual address of the pagetable entry. In other words, a traditional scheme would traverse the tables in Figure 2.1 from right to left, while in-cache translation traverses them from left to right.

The SPUR implementation simplifies the translation process slightly by stopping the translation at the root pagetable. The operating system locks down the root pagetable pages at "well-known" locations in physical memory. This makes it possible to skip explicit accesses to the root[2] and root[3] pagetables. The next two sections describe cache accesses and the translation algorithm.

## 2.1. Virtual Address Cache Accesses

The cache is accessed using global virtual addresses (GVA), so the translation from process virtual addresses (PVA) must be done first. The top two bits of the process virtual address select

one of the four active segments, which are specified by four *global segment number (GSN)* registers on the cache controller. The 8-bit global segment number is read out of the selected GSN register, and prepended to the remaining 30 bits of the process virtual address to form the 38-bit global virtual address. This process is illustrated in Figure 2.2 and section A of Figure 2.3. Because the segments are large, the cache line can be accessed in parallel with the GSN lookup; the high-order bits of the GVA are only needed during tag comparison.

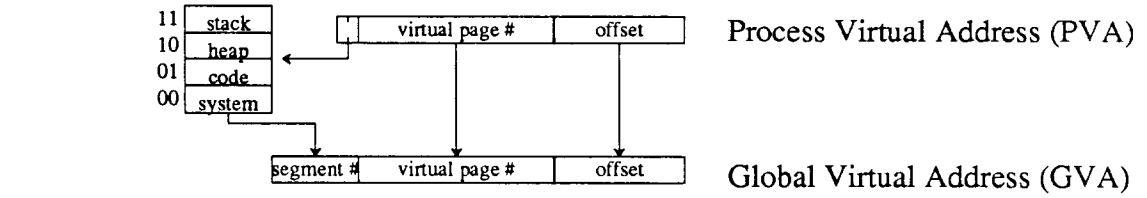


Figure 2.2: Formation of Global Virtual Address

If the tags match, then the associated status bits are checked to make sure the access may proceed. The *coherency state* has four values: Invalid, UnOwned, OwnShared, and OwnPrivate. A processor read is permitted if the state is UnOwned, OwnShared, or OwnPrivate. A processor write can proceed without access to the system bus only if the state is OwnPrivate and the *PageDirty* bit is set (see Section 2.3.2). Finally, the protection is checked to verify access permission (see Section 2.3.1).

## 2.2. Translation Algorithm

The translation algorithm is implemented in hardware. All exceptional conditions, such as page faults, are detected in hardware, which aborts the current instruction and invokes a software trap handler. This section describes the translation algorithm. The registers used during address translation are accessible via control space, and are discussed in Section 4.

When a processor reference hits in the cache, no translation is necessary. However, when there is a cache miss, we need to access the pagetable entry (PTE) to perform translation. Because the cache also serves as a translation buffer, we look there for the PTE. First, we compute the global virtual address of the PTE by using the virtual page number of the GVA (bits 37:12) as an index into the pagetable. To simplify the computation, the pagetable is required to be contiguous, and aligned on a 256 megabyte boundary in virtual space; the base address of the pagetable is maintained in the PTEVA register. The address computation is simply to shift the GVA right 10 bits and concatenate with the top 10 bits of the PTEVA; in the implementation, the shifted GVA is stored into the low-order bits of the PTEVA. The PTEVA computation is illustrated in step B of Figure 2.3.

The PTEVA now contains the global virtual address of the PTE, which we use to access the cache. If the PTE is in the cache, we check the PTEPageValid, PTEPageReferenced, and PTEPageCacheable status bits (described below) to see if we can bring the data into the cache. If

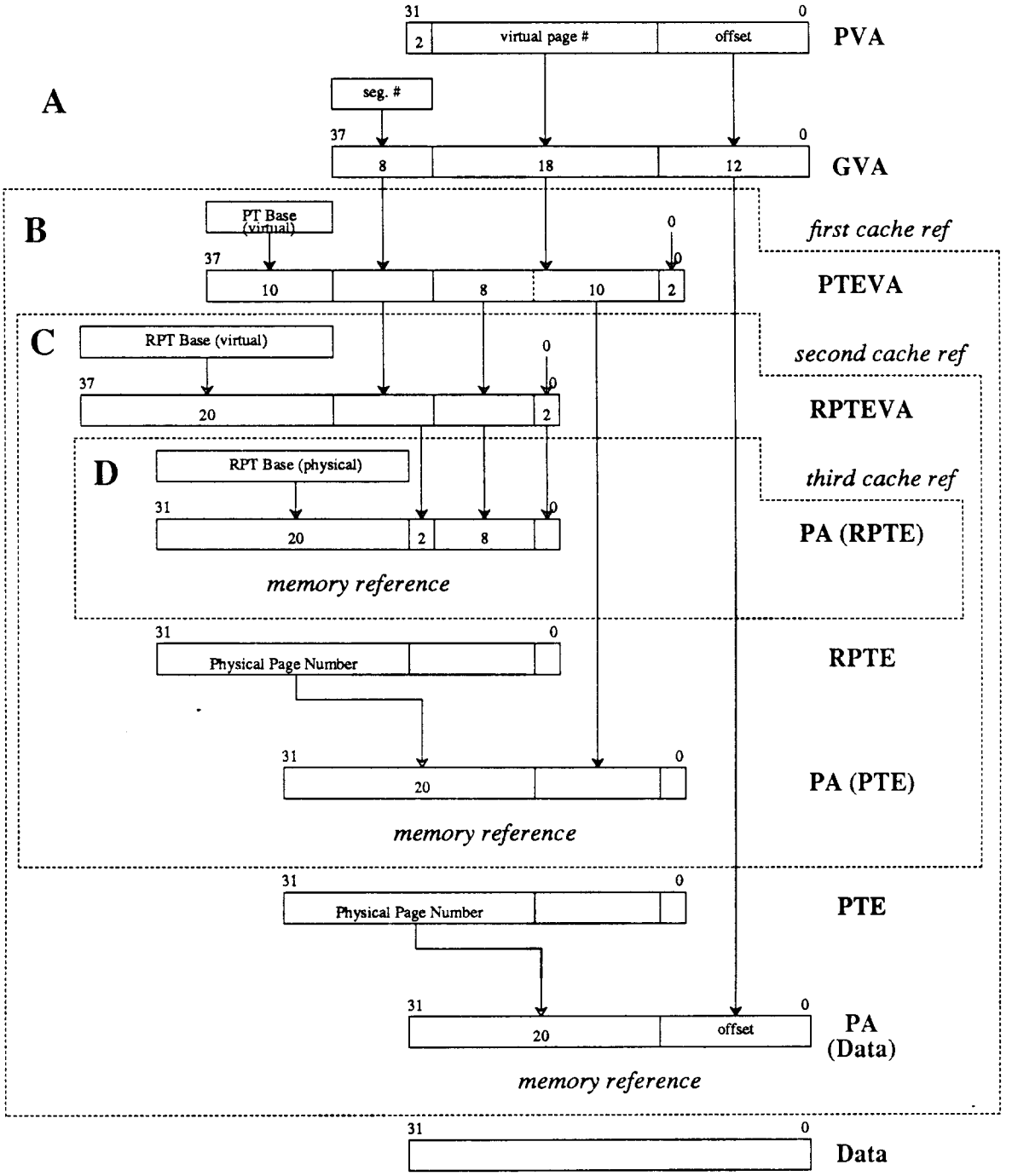


Figure 2.3: Translation from Virtual to Physical Addresses

so, we extract the physical page number and form the data's physical address, then transfer the block into the cache to complete the miss handling. If the PTE is not in the cache, a fairly infrequent occurrence, then we have to find the root pagetable entry (RPTE) to obtain the physical address of the PTE. Logically, we perform the same shift and concatenate operation to compute the virtual address of the RPTE (the RPTEVA), but this time start with the PTEVA instead of the GVA. However, to simplify the implementation the actual computation is a little different. The high-order 20 bits of the RPTEVA register contain the base address of the root pagetable in virtual space. The PTEVA is shifted right 10 bits, and the low-order 18 bits are stored into the low-order bits of the RPTEVA. This process is illustrated in step C of Figure 2.3.

We now use the RPTEVA to look for the RPTE in the cache. If we find the RPTE in the cache, then we check only the PTEPageValid bit. The PTEPageReferenced and PTEPageCacheable bits are ignored. We then use the physical page number to bring the PTE into the cache, and proceed as above. If the RPTE is not in the cache, we have to locate it in physical memory. Rather than shift and concatenate again, we maintain four *root pagetable map* (RPTM) registers that point into the root pagetable in physical memory. The registers point to the regions corresponding to the four active segments. The physical address of the RPTE is computed by simply concatenating the base physical address from the appropriate RPTM with the page offset from the virtual address (in the RPTEVA). This is illustrated in step D of Figure 2.3.

The operating system is responsible for maintaining correct information in the address translation registers. This is discussed further in Section 4.

## 2.3. Pagetable Entry Format

Each pagetable entry contains a 20 bit physical page number (PTEPagePhysicalAddress), two bits of protection (PTEPageProtection), and 5 status bits that are understood by the hardware (PTEPageValid, PTEPageDirty, PTEPageReferenced, PTEPageCacheable, and PTEPageCoherency). The PTE format is described in Table 2.1 and illustrated in Figure 2.4. The PTEPageValid bit indicates that the page is resident in main memory, in the physical pageframe

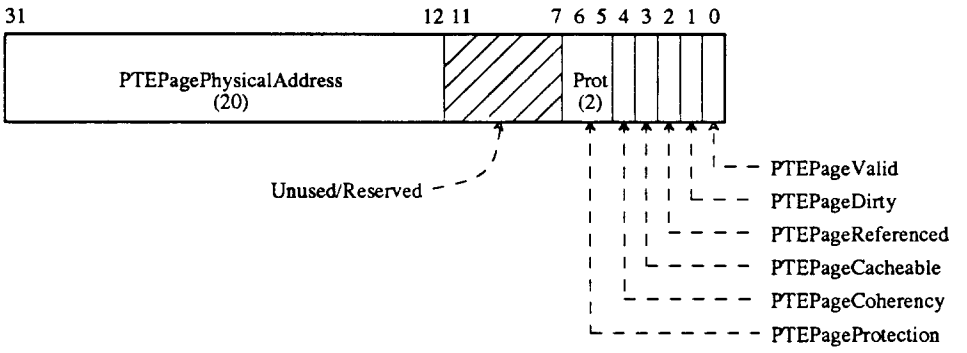


Figure 2.4: Pagetable Entry (PTE) Format

| Table 2.1: PageTable Entry Format | |
|---|---|
| PTE Field | Description |
| PTEPageValid (bit 0) | Indicates that the page is valid. |
| PTEPageDirty (bit 1) | Indicates that the page is dirty. |
| PTEPageReferenced (bit 2) | Indicates that the page has been referenced. If the page is marked as unreferenced, then the true status (referenced or unreferenced) of the page is uncertain. |
| PTEPageCacheable (bit 3) | Indicates that the page is a cacheable page. |
| PTEPageCoherency (bit 4) | When a miss occurs, the PTEPageCoherency bit causes the cache controller to fetch the block with ownership, regardless of the reference made by the CPU. This is an important parameter for tuning the cache behavior. |
| PTEPageProtection (bits 6:5) | Used to hold the protection bits of the page. The protection values are described below. |
| PTEPageReserved (bits 11:7) | 5 bits that are undefined in the architecture. Use at your own risk. |
| PTEPagePhysicalAddress (bits 31:12) | Physical page number for this page, if the page is valid. |

addressed by PTEPagePhysicalAddress. If the page is not valid, then PTEPagePhysicalAddress is undefined (may be used by software for any other purpose).

PTEPageCoherency is a hint to the cache controller that the page is probably not shared, and that on a cache miss it should fetch blocks with ownership. PTEPageCoherency only applies when referencing data and instructions; the cache controller always fetches PTEs as shared regardless of the value. Since PTEs are usually modified when accessed as data, the PTEPageCoherency bit should probably be set in the root pagetable entries (although it need not).

PTEPageCacheable indicates that the hardware may cache blocks from the page. Loads and stores to non-cacheable pages cause single-word bus reads and writes, bypassing the cache completely. This allows virtual memory access to I/O device control registers. Note that the operating system *must* make the pagetables cacheable (i.e., PTEPageCacheable must be set in root pagetable entries), since the hardware does not check the PTEPageCacheable bit when fetching PTEs.

### 2.3.1. Protection

Protection is provided on a page basis, i.e., each page may have a different protection mode. When a block is brought into the cache, the protection is copied from the PTE into the cache line. When the processor references a cache line, the cached protection bits are checked for access permission. Because of this caching, changing the protection in a PTE does not necessarily affect all blocks in the associated page immediately. A selective cache flush is necessary to propagate the

update.

SPUR provides four protection modes, listed in Table 2.2. Since there is no distinction between reads and instruction fetches, all code pages must be readable.

When the translation recursion terminates (when the cache controller misses on an RPTE) there is no PTE to specify the correct protection. Instead, the cache controller forces the protection to be PROT_KRW_UNA.

### 2.3.2. Reference and Dirty Bits

Most systems maintain a reference bit for each page to improve the performance of their page replacement algorithm. Similarly, they also maintain a dirty bit to eliminate unnecessary writes to secondary storage. In SPUR, these bits are called *PTEPageReferenced* and *PTEPageDirty*. However, because of the virtual address cache, these bits are handled slightly differently. PTEPageReferenced differs from the reference bit of other systems because it is an approximation, rather than being completely accurate. PTEPageReferenced is set only when a cache miss occurs to that page; more specifically, if PTEPageReferenced is not set when a miss occurs, the cache controller generates a fault and sets the Reference Fault bit in the fault register (see Section 5.2). The trap handler is then responsible for setting the PTEPageReferenced bit. The PTEPageReferenced bit is only checked when accessing data and instructions, not pagetable entries. If the pagetable is actually paged (most operating systems don't do this) then a separate mechanism is needed to select a pagetable page as a victim. Note that the cache controller assumes that if a data page is valid, so is its PTE. Thus a pagetable page can only be paged out if none of the pages it maps are valid.

A similar strategy is employed for dirty bits. When a block is brought into the cache, the PTEPageDirty bit is copied from the PTE into the cache line's PageDirty state bit. When the processor initiates a write, the PageDirty bit is checked. If it is 1, the write proceeds normally. If not, the write is handled as a miss, and the PTE re-examined. If the PTEPageDirty bit is 1, then the miss handling is completed (the PageDirty bit is automatically updated), and the write allowed to proceed. If PTEPageDirty is 0, however, the cache controller generates a DirtyBit Fault. The trap handler is then responsible for updating PTEPageDirty.

When a page is flushed out to disk, care must be taken to insure consistency between the the cache and the PTE. This topic is discussed further in Section 7.

**Table 2.2: Protection Modes**

| Protection Code | Value | Description |
|---|---|---|
| PROT_KRO_UNA | 0 | Kernel has read-only access, user has no access. |
| PROT_KRW_UNA | 1 | Kernel has read-write access, user has no access. |
| PROT_KRW_URO | 2 | Kernel has read-write access, user has read-only access. |
| PROT_KRW_URW | 3 | Kernel has read-write access, user has read-write access. |

## 3. Physical Address Space

The physical address space is primarily accessed by **ld_32** and **st_32** while in physical mode; that is, when the K_VirtDFetch bit is not set in the KPSW. In general, this should only occur while the CPU is in kernel mode, because it can be used to bypass all other protection mechanisms. It is also possible to access some parts of the physical address space using non-cacheable pages. **Ld_32** and **st_32** to non-cacheable pages behave normally, except that they completely bypass the cache. This feature is provided to simplify access to external I/O devices. Since protection is supported on a page basis, it is possible to allow users access to some devices, e.g., a bitmap display, while restricting access to others.

The physical address space of a SPUR processor, shown in Figure 3.1, is composed of an on-board local space and a global external space. The local space is further divided into addresses for an EPROM, a UART, status LEDs and direct access to the cache. The global space is divided into per board slot spaces and a large region left for software definition.

### 3.1. EPROM

The 16K words of local EPROM are designed for use by debug, diagnostics and bootstrap utilities. The EPROM provides a safe place to execute code when the world starts to dissolve; it allows us to avoid the infamous "double bus-error" problem of the MC68000.

The EPROM address space starts at physical address 0x00000000 and extends upwards to 0x0000FFFF.

### 3.2. UART

The UART is the primary communication link with the outside world during system debug. It allows a remote hosted debugger to communicate with the CPU without relying on the SpurBus. The UART found on the SPUR processor board is a Signetics SCN68681 Dual Universal Asynchronous Receiver/Transmitter (DUART) [Sign83]. This device provides 2 independent UART interfaces, each with programmable baud rate, optional flow control, and polled or interrupt driven operation. It also provides a single interrupting counter/timer (C/T). The transmitter of each channel has single character buffering and the receiver has a 3 character FIFO. An interrupt can be generated by either channel under program control (via a mask) when break is either detected or ceases, when the receive FIFO has any characters in it or is full, when the transmitter is prepared to accept a character, or when the counter/timer reaches zero.

The DUART is accessed in the physical address space beginning at address 0x00010000. Its registers are listed with their respective offsets in Table 3.1. Each register is a single byte wide aligned on a word boundary. For a full discussion of DUART programming see the Signetics data sheet. Below is a brief description of important registers and some recommended control values.

#### 3.2.1. Mode Registers (MR1A, MR2A, MR1B, MR2B)

Each channel has 2 different mode registers located at one address. The first access to a mode register will go to MR1x. Subsequent accesses all go to MR2x until the "reset mode register pointer" command (0x10) is written to the corresponding command register. Do not change a mode register while the corresponding receiver or transmitter is enabled.

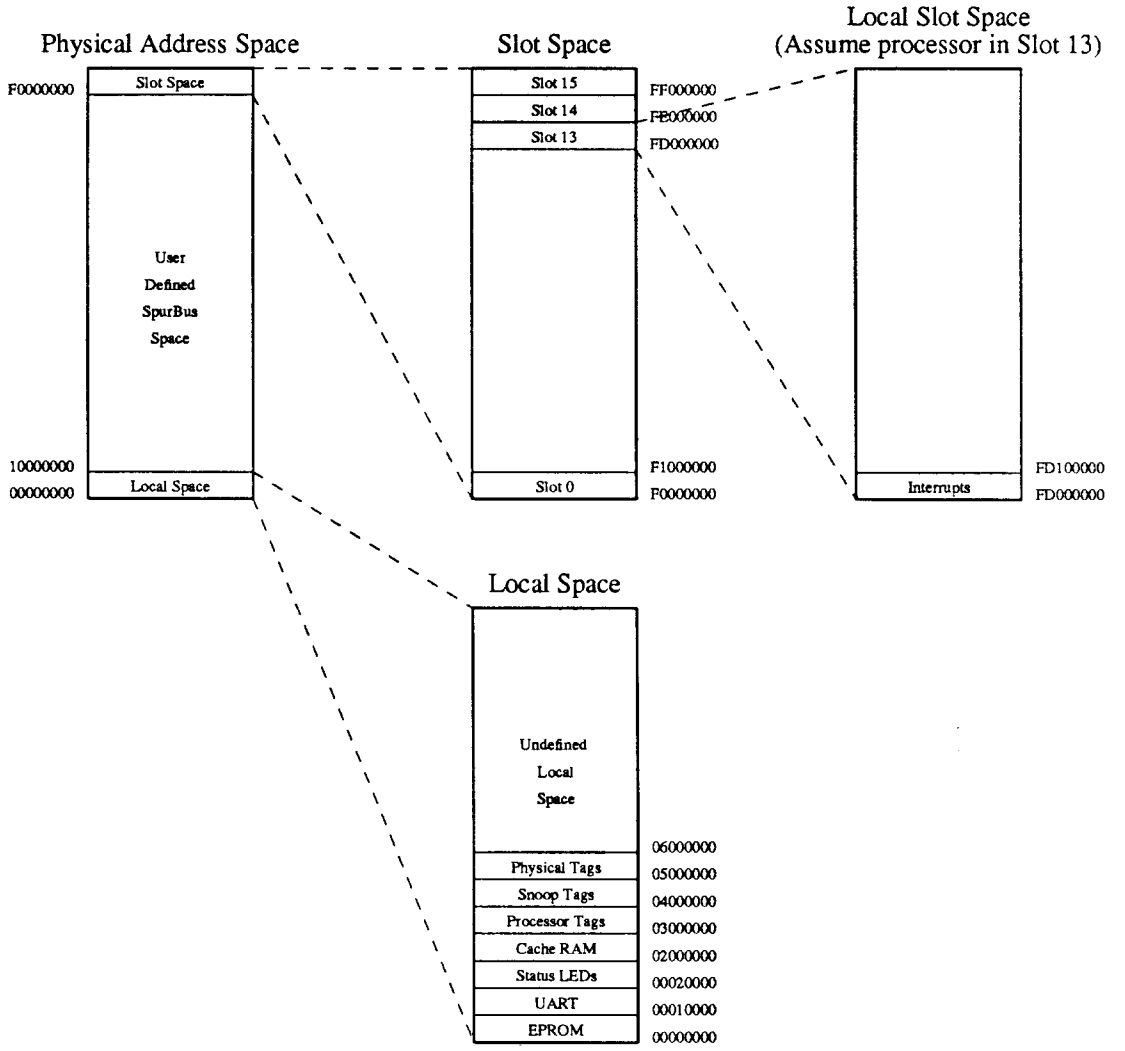


Figure 3.1: SpurBus Physical Address Space Organization

| Table 3.1: DUART Register Addressing | | | | |
|---|---|---|---|---|
| Offset from 0x00010000 | CPU operation ld_32 (physical) | | CPU operation st_32 (physical) | |
| Channel A | | | | |
| 00 | mode register | MR1A,MR2A | mode register | MR1A,MR2A |
| 04 | status register | SRA | baud rate register | CSRA |
| 08 | reserved | | command register | CRA |
| 0C | receive FIFO | RHRA | transmit buffer | THRA |
| Channel B | | | | |
| 20 | mode register | MR1B,MR2B | mode register | MR1B, MR2B |
| 24 | status register | SRB | baud rate register | CSRB |
| 28 | reserved | | command register | CRB |
| 2C | receive FIFO | RHRB | transmit buffer | THRB |
| Miscellaneous | | | | |
| 10 | input port change register | IPCR | auxiliary control register | ACR |
| 14 | interrupt status register | ISR | interrupt mask register | IMR |
| 18 | C/T high order byte | CTU | C/T high order byte | CTUR |
| 1C | C/T low order byte | CTL | C/T low order byte | CTLR |
| 30 | interrupt vector register | IVR | interrupt vector register | IVR |
| 34 | input port register | IP | output port configuration | OPCR |
| 38 | start C/T command | | set output port bits command | |
| 3C | stop C/T command | | reset output port bits command | |

Load MR1 with 0x13 to get 8 bit data transfers with no parity, no flow control, receive interrupts whenever the receive FIFO is not empty (when these interrupts are enabled) and error status reporting for the byte at the top of the FIFO (instead of reporting the logical OR of errors in each FIFO position). For even parity use 0x03, for seven bit data use 0x12, for interrupts only when the FIFO is full use 0x53 and for flow control[1] use 0x93.

Load MR2 with 0x07 to get a single stop bit and ignore flow control from the remote end of a connected cable. To put a channel into loop-back mode, use 0x87, to get 2 stop bits use 0x0F, and for remote flow control use 0x17.

### 3.2.2. Baud Rate Register (CSR)

Load this write only register with 0xBB to transmit and receive at 9600 baud. Other values are 0x11 for 110 baud, 0x44 for 300 baud, 0x66 for 1200 baud, 0x88 for 2400 baud, 0x99 for 4800 baud and 0xCC for 19.2K baud. Do not change this register while the corresponding receiver or transmitter is enabled.

[1] SPRITE on SUN hardware does not use flow control; however, for higher performance down-loading, this may become useful. RTS (ready to send) and CTS (clear to send) are used for flow control. If RTS on this end is connected to CTS on the other end and if the other end is enabled to inhibit transmitting unless it sees CTS asserted, then the above setting for flow control will cause the concerned DUART channel to halt transmission from the remote to itself whenever its FIFO is full and will resume transmission as soon as the first byte is read out of the FIFO. To use flow control, software must also set OP0 and OP1 after each hardware reset (by writing 0x03 to 0x00010038).

| Table 3.2: Command Register (CR) | | |
|---|---|---|
| Field | Value | Command |
| 1-0 | 0 | noop |
| | 1 | enable receive |
| | 2 | disable receive |
| 3-2 | 0 | noop |
| | 1 | enable transmit |
| | 2 | disable transmit |
| 7-4 | 0 | noop |
| | 1 | reset mode register pointer to MR1 |
| | 2 | reset receiver state |
| | 3 | reset transmitter state |
| | 4 | clear status bits that report error conditions in SR |
| | 5 | clear break state change interrupt bit in ISR and SR |
| | 6 | start transmitting break |
| | 7 | stop transmitting break |

### 3.2.3. Command Register (CR)

This write only register is composed of three independently interpreted fields: receive, transmit and miscellaneous as described in Table 3.2.

### 3.2.4. Status Register (SR)

The status register is read only and is shown in Table 3.3. This register should be read before each byte is read out of the FIFO. Note that the error reporting bits can apply either to the topmost byte in the FIFO or to the logical OR of all error bits in the FIFO according as determined by bit 5 in MR1. We recommend the former.

| Table 3.3: Status Register (SR) | |
|---|---|
| Bit | Set interpretation |
| 7 | break has been received |
| 6 | framing error (stop bit missing) |
| 5 | parity error |
| 4 | FIFO overrun |
| 3 | transmitter empty (TxEMT) |
| 2 | transmitter ready for another byte (TxRDY) |
| 1 | receiver FIFO full (RxFFUL) |
| 0 | receiver FIFO not empty (RxRDY) |

### 3.2.5. Interrupt Status Register (ISR)

The DUART generates an interrupt (that can be masked in the cache controller register IMask) whenever the logical AND of ISR and IMR is non-zero. The bits of this register are described in Table 3.4.

### 3.2.6. Interrupt Mask Register (IMR)

The bits of the interrupt mask register correspond to the bits of the interrupt status register. If a bit is set in the mask, the corresponding interrupt is enabled.

### 3.2.7. Interrupt Vector Register (IVR)

On reset, the interrupt vector register is set to 0x0F by the DUART, otherwise, it is managed by CPU software. This may not be very useful.

### 3.2.8. Input Port Change Register (IPCR)

The Signetics DUART has 6 general use input pins, IP0 - IP5. This register provides, for each of IP0 - IP3, the current value and a bit indicating at least one change since the last processor read. However, in SPUR, bits IP2 - IP5 are not connected and IP0, IP1 are connected to the remote CTSA and CTSB. So this register is of little use; still, IPCR<7:4> are set as IP3 - IP0 are changed and IPCR<3:0> have the current values of IP3 - IP0.

Table 3.4: Interrupt Status Register (ISR)

| Bit | Description |
|---|---|
| 7 | changes in IP3-IP0 (see ACR)<br>reset by reading IPCR |
| 6 | change in channel B break status<br>reset by clear break state change interrupt command in CR |
| 5 | channel B FIFO interrupt (see MR1)<br>reset by reading channel B FIFO |
| 4 | channel B transmitter ready for another byte<br>reset by writing to transmitter |
| 3 | counter/timer has 'rolled over'<br>read the 'stop C/T command' register |
| 2 | change in channel A break status<br>reset by clear break state change interrupt command in CR |
| 1 | channel A FIFO interrupt (see MR1)<br>reset by reading channel A FIFO |
| 0 | channel A transmitter ready for another byte<br>reset by writing to transmitter |

| Table 3.5: Auxiliary Control Register (ACR) | | |
|---|---|---|
| Field | Value | Command |
| 7 | 1 | sets the maximum baud rate to 19.2K |
| 6-4 | 000 | counter clock from IP2 (not connected in SPUR) |
| | 001 | counter clock from channel A transmission baud rate |
| | 010 | counter clock from channel B transmission baud rate |
| | 011 | counter clock frequency 3.6864 /16 MHz |
| | 100 | timer clock from IP2 (not connected in SPUR) |
| | 101 | timer clock from IP2 /16 (not connected in SPUR) |
| | 110 | timer clock frequency 3.6864 MHz |
| | 111 | timer clock frequency 3.6864 /16 MHz |
| 3-0 | | if bit n is set, then a change on IPn will set ISR<7> |

### 3.2.9. Auxiliary Control Register (ACR)

This miscellaneous write only control register should be loaded with 0x80. If a free running timer with a known absolute time frequency is important, load 0xE0. This will cause the timer to set ISR<3> every (CTUR,CTLR)*542.5 nsec. ISR<3> is cleared by reading the 'stop C/T command' register. See the Signetics data sheet for counter use.

### 3.2.10. Output Port Configuration Register (OPCR)

In SPUR most of the input and output port bits are not connected. IP0 and IP1 are used to receive CTSA and CTSB from remote end. OP0 and OP1 are used to generate RTSA and RTSB to the remote end. The intention is to allow for flow control if desired. However, the other DUART input and output ports provide a relatively simple interface for software control of SPUR processor board hardware modifications. Software can read the input port bits and can individually set or reset the output port bits.

The output pins OP2 - OP7 can be configured to be software controlled by loading 0x00 into OPCR. See Signetics the data sheet for other configurations.

## 3.3. Status LEDs and Seven Segment Displays

During system debug and and after system failures we need processor state information to diagnose low-level problems. The UART is the main interface for obtaining this state; however, each processor board also has seven light emitting diodes (LEDs) and two seven segment displays (the familiar digital watch numeric format). Three of the LEDs are not under processor control, and display important implementation level state. The other four LEDs and the seven segments displays are controlled by the Display Register.

A **st_32** to any physical address between 0x00020000 and 0x0002FFFF updates the Display Register. The low-order 15 bits of this register determine the display pattern as described in Tables 3.6 and 3.7. This register can only be written; the result of reads is undefined.

| Table 3.6: Display Register | |
|---|---|
| Field | Command |
| 0-3 | least significant digit segment specifier |
| 4-7 | most significant digit segment specifier |
| 8 | light status LED 0 |
| 9 | light status LED 1 |
| 10 | light status LED 2 |
| 11 | light status LED 3 |
| 12 | lamp test (force all segments on) |
| 13 | light least significant digit decimal point |
| 14 | light most significant digit decimal point |

| Table 3.7: Seven Segment Display Digit Specifier | |
|---|---|
| Value | Segment Display |
| 0,1,2,3,4,5,6,7,8,9 | 0,1,2,3,4,5,6,7,8,9 |
| A | 'small c' |
| B | 'backwards small c' |
| C | 'raised small u' |
| D | 'capital E missing the lower half of the vertical line' |
| E | 'capital E missing the top horizontal line' |

### 3.4. Cache Rams

SPUR's performance depends on its cache memory's ability to satisfy loads, stores and instruction fetches without accessing main memory. The cache relies on the principle of locality: processor references to main memory tend to cluster in small regions of the address space for extended periods of time. The cache saves copies of these regions in fast static rams. When the processor makes a reference to memory, the cache checks to see if it has that data locally and, if so, provides it without accessing main memory.

SPUR's cache is 128 Kbytes, direct-mapped, with instructions and data mixed together, 32 byte blocks, and delays memory updates as long as possible (writeback).

The cache memory is composed of data, tag, and state rams. The (virtual) tag rams contain the high-order bits of the global virtual address (bits 37:17) for each block of cached data. The state rams hold 4 bits for each cache block, to describe the current status. Two of these bits describe the *coherency state*: Invalid, UnOwned, OwnShared, or OwnPrivate. A processor read will hit in the cache if the state is UnOwned, OwnShared, or OwnPrivate (and the tag matches, of course). A processor write will only hit if the state is OwnPrivate. In addition, the *PageDirty* bit must also be set for a write to proceed without accessing the main memory (see Section 2.3.2). The fourth state bit, *BlockDirty*, is set whenever a block is modified. When a block is about to be replaced, if the *BlockDirty* is set, it is first written back to memory.

To support the cache coherency protocol (see Section 6), a second copy of the tags and state is provided. This copy allows the bus watching (snooping) operations to take place without unnecessary interference with the processor. Only the coherency state is duplicated, since the two dirty bits are not used by the coherency protocol.

The SPUR cache supports three access units: 32 bits, 40 bits and 64 bits. To accommodate all of these in a single access, the data rams are organized 64 bits wide. Sequential 32 bit accesses alternate between the "high" and "low" halves of the data rams.

As described in Section 2, a virtual address is translated to a physical address on cache misses. To avoid redoing this translation on writebacks, the physical address is saved in the *physical tag* ram. Only a single copy of this tag is required.

### 3.4.1. Cache Data Rams

When addressed via the physical address space, the cache data rams behave as normal memory, without the associative behavior of a cache. The cache ram addresses begin at 0x02000000. The cache rams are only 32K words, so bits 17 through 23 are ignored and the addresses wrap around.

During bootstrap and debug, the cache can be used as a 128KB scratch RAM. This can facilitate some forms of diagnostics.

### 3.4.2. State and Tag Rams

The processor's copy of the cache state and tags may be accessed at any time, using the **ld_st** and **st_32** instructions in physical mode. These rams are addressed starting at 0x03000000. Bits 16 through 5 of the address are used to select the block; note that these are not the least significant bits. The 21 bit virtual tag (corresponding to bits 37 through 17 of the cache block's global virtual address) is returned in bits 31 through 11 of the data word. In this same returned data word is: the 2 bit protection field (see Section 2.3.1) in bits 6 and 5, the PageDirty bit in bit 3, the BlockDirty bit in bit 2, and the coherency state on bits 1 and 0. Bits 10 through 7, and 4 are undefined. Storing to these rams is done in the same way, with the same bit assignments in the data word.

The snooping copy of the cache state and tags is identical to the processor's copy, except that it is addressed beginning at 0x04000000 and neither the 2 bit protection code nor the dirty bits are maintained. There is also a constraint on when the processor can access the snoop rams; specifically, coherency bus watching should not occur while these rams are being read or written. This is necessary to prevent collisions between the processor and snoop sides of the cache controller, which would cause indeterminate results. Preventing snooping events from occurring must be handled in software, and should be reserved for diagnostics and self-test.

### 3.4.3. Physical Address Tags

The cache controller maintains one copy of the physical address tag for each block in the cache. These tags can be accessed starting at address 0x05000000. Bits 16 through 5 are used to select the physical address tag. The 20 bit physical tag appears in bits 31 through 12 of the data word (corresponding to bits 31 through 12 of the cache block's physical address), with bits 11 through 0 undefined.

### 3.5. SpurBus Space

Main memory and all peripheral devices are located in the physical address space and accessed via the SpurBus[2]. Each port onto the SpurBus (typically linking a SPUR board, memory board or I/O controller to the SpurBus) is called a *slot* in NuBus jargon. Each slot identifies itself by a unique, backplane-provided *slot id* as described in Section 4.4. In this way board 0 is defined to be the board in slot 0.

As shown in Figure 3.1, the NuBus divides the 4 Gbyte physical address space into a 256 Mbyte *slot space* starting at 0xF0000000 and a 3.75 Gbyte *user space* everywhere else. Slot space is then subdivided into 16 pieces each 16 Mbytes. Each slot on the NuBus owns one of these pieces which we call the slot's *local slot space*. The lowest address of board j's local slot space is 0xFj000000. Notice that the local space (EPROM, UART, etc) of all processors overlap; a processor cannot address another processor's local space; but, each processor can address any board's local slot space.

The NuBus requires a *slot occupancy* test to be satisfied by every NuBus board: when the highest order word of its local slot space (at address 0xFjFFFFFC for board j) is read, the board must respond (with a successful or failed status). All SPUR board's pass this test by responding to single word reads or writes at this address with a bus error status (see Section 5).

The NuBus also requires a *configuration ROM* located in the high addresses of each board's local slot space. Unfortunately SPUR processor board's do not provide a configuration ROM.

The most common use of slot space is for inter-processor communication. In the general case each NuBus board maps its local slot space in an arbitrary fashion[3]. SPUR processor boards provide only one form of mapped communications: 16 dataless interrupts. To generate interrupt i on board j, execute a single word write to physical address:

$$0xF<j>0XXX<xx i_3 i_2><i_1 i_0 00>$$

Notice that bits 19:6 of the address are ignored (we recommend using zeros) and that a data word will be transmitted and ignored.

In summary, SPUR processor boards respond to all reads into their local slot space and all writes into the top 15 sixteenths of their local slot space with a bus error. They respond to all writes into the bottom sixteenth of their local slot space with successful status and treat these writes as dataless interrupts.

---

[2] A complete description of the SpurBus can be found in the SpurBus Specification manual [Gibs85].

[3] SPUR systems are expected to use Texas Instruments NuBus boards for main memory, monitor, network interface, disk controllers, etc.

## 4. Control Space

The control space provides access to the cache controller registers and functions. The cache controller registers are used by the hardware during address translation and error handling. Control space accesses can only be made using the **ld_external** and **st_external** instructions. Bits 4 through 2 of of the effective address are interpreted as a *subcacheop*, specifying the request. The prototype supports 4 subcacheops, listed in Table 4.1. Since these are privileged instructions, their use in user mode will cause an access violation.

**RESET** generates a bus reset. The issuing processor returns with normal status, all other processors handle the reset as an error (see Section 5.3). Since errors can result in lost state, RESET should be used sparingly.

**Table 4.1: SubCacheOp Description**

| SubCacheOp | | Description |
|---|---|---|
| Name | Value | |
| RESET | 1 | Generate a NuBus Bus Reset. Returns normally on the issuing processor. |
| RDREG | 2 | Read the cache register byte specified by effective address bits 16:5. Legal only with **ld_external**. |
| WRREG | 3 | Write the cache register byte specified by effective address bits 16:5. Legal only with **st_external**. |
| FLUSH | 4 | Flush the block specified by bits 16:5 of the effective address from the cache. |

The **FLUSH** operation purges a cache line. The bits 16 through 5 of the effective address specify the cache line to be purged. Owned blocks are written back to memory, and the cache state is set to Invalid. Both **RESET** and **FLUSH** can be used with either **ld_external** or **st_external**.

**ld_external**, with the subcacheop **RDREG**, reads a byte from the cache controller register specified by the high-order bits of the effective address. The byte is loaded into the low-order byte of the destination register. Similarly, **st_external**, with the subcacheop **WRREG**, stores the low-order byte from the source register to the specified byte in the cache controller.

Because some of the registers are longer than 32-bits, and because of pin limitations in the prototype implementation, only one byte is transferred on each load or store. The effective address of the **ld_external** instruction specifies the register and the byte within the register, in addition to the 3-bit subcacheop. Bits 12 through 8 specify the register, and 7 through 5 specify the byte within the register; three bits are used to specify the byte since some registers are 38 bits. Bits 4 though 2 specify the subcacheop (see Table 4.1). Table 4.2 lists the addresses for each byte in each cache register.

| Table 4.2: Cache Controller Register Addresses | | | | | |
|---|---|---|---|---|---|
| Address<12:8> | Address<7:5> | | | | |
| | 100 | 011 | 010 | 001 | 000 |
| 00000 | GSN0<7:0> | RPTM0<19:12> | RPTM0<11:04> | RPTM0<03:00> | --- |
| 00001 | GSN1<7:0> | RPTM1<19:12> | RPTM1<11:04> | RPTM1<03:00> | --- |
| 00010 | GSN2<7:0> | RPTM2<19:12> | RPTM2<11:04> | RPTM2<03:00> | --- |
| 00011 | GSN3<7:0> | RPTM3<19:12> | RPTM3<11:04> | RPTM3<03:00> | --- |
| 00100 | GVA<37:32> | GVA<31:24> | GVA<23:16> | GVA<15:8> | GVA<7:0> |
| 00101 | PTEVA<37:32> | PTEVA<31:24> | PTEVA<23:16> | PTEVA<15:8> | PTEVA<7:0> |
| 00110 | RPTEVA<37:32> | RPTEVA<31:24> | RPTEVA<23:16> | RPTEVA<15:8> | RPTEVA<7:0> |
| 00111 | G<37:32> | G<31:24> | G<23:16> | G<15:8> | G<7:0> |
| 01000 | --- | T0<31:24> | T0<23:16> | T0<15:8> | T0<7:0> |
| 01001 | --- | T0<63:56> | T0<55:48> | T0<47:40> | T0<39:32> |
| 01010 | --- | T1<31:24> | T1<23:16> | T1<15:8> | T1<7:0> |
| 01011 | --- | T2<31:24> | T2<23:16> | T2<15:8> | T2<7:0> |
| 01100 | --- | IStatus<31:24> | IStatus<23:16> | IStatus<15:8> | IStatus<7:0> |
| 01101 | --- | IMask<31:24> | IMask<23:16> | IMask<15:8> | IMask<7:0> |
| 01110 | --- | FEStatus<31:24> | FEStatus<23:16> | FEStatus<15:8> | FEStatus<7:0> |
| 01111 | --- | --- | --- | Mode<7:0> | SlotId<3:0> |
| 10000 | --- | C0<31:24> | C0<23:16> | C0<15:8> | C0<7:0> |
| 10001 | --- | C1<31:24> | C1<23:16> | C1<15:8> | C1<7:0> |
| 10010 | --- | C2<31:24> | C2<23:16> | C2<15:8> | C2<7:0> |
| 10011 | --- | C3<31:24> | C3<23:16> | C3<15:8> | C3<7:0> |
| 10100 | --- | C4<31:24> | C4<23:16> | C4<15:8> | C4<7:0> |
| 10101 | --- | C5<31:24> | C5<23:16> | C5<15:8> | C5<7:0> |
| 10110 | --- | C6<31:24> | C6<23:16> | C6<15:8> | C6<7:0> |
| 10111 | --- | C7<31:24> | C7<23:16> | C7<15:8> | C7<7:0> |
| 11000 | --- | C8<31:24> | C8<23:16> | C8<15:8> | C8<7:0> |
| 11001 | --- | C9<31:24> | C9<23:16> | C9<15:8> | C9<7:0> |
| 11010 | --- | C10<31:24> | C10<23:16> | C10<15:8> | C10<7:0> |
| 11011 | --- | C11<31:24> | C11<23:16> | C11<15:8> | C11<7:0> |
| 11100 | --- | C12<31:24> | C12<23:16> | C12<15:8> | C12<7:0> |
| 11101 | --- | C13<31:24> | C13<23:16> | C13<15:8> | C13<7:0> |
| 11110 | --- | C14<31:24> | C14<23:16> | C14<15:8> | C14<7:0> |
| 11111 | --- | C15<31:24> | C15<23:16> | C15<15:8> | C15<7:0> |

## 4.1. Address Translation Registers

A number of registers are required to support the in-cache address translation algorithm. The general translation mechanism is described in Section 2.1, but the details of the registers are described here.

### 4.1.1. GSN Registers

The global segment number (GSN) registers, also known as the active segment registers, maintain the principle mapping from process virtual addresses to global virtual addresses. These four registers, one for each of the four active segments, contain the 8-bit global segment numbers used to generate the global virtual address. The top two bits of the 32-bit process virtual address select one of the four active segments. The 8-bit global segment number is read out of the

selected register and prepended to the remaining 30 bits of the processor virtual address to form the 38-bit global virtual address. The four segments are usually assigned to system code and data, user code, private data (e.g., stack and private heap), and shared data.

### 4.1.2. PTEVA Register

The pagetable entry virtual address (PTEVA) register is a 38-bit special register. It can be read and written in a byte-wise manner, like all other cache controller registers, but the low-order 28 bits are set by hardware each time a processor reference occurs. The high-order 10 bits should be set to contain the high-order 10 bits of the global virtual address of the pagetable. Since writes to the register occur a byte at a time, setting the high-order 10 bits corrupts the low-order 28. The diagnostic programmer should be aware of this behavior. On a processor reference, the low-order bits of the PTEVA (bits 27:2) are set to the virtual pagenumber for the data requested by the CPU, that is, bits 37:12 of the GVA. The low-order two bits are set to zero. If the reference misses in the cache, the cache controller then uses this address to reference the pagetable entry.

### 4.1.3. RPTEVA Register

The root pagetable virtual address (RPTEVA) register is similar to the PTEVA register. It is also 38-bits wide, but differs in that the high-order 20 bits are the high-order address bits of the root pagetable. Note that these top 20 bits should generally be two copies of the top 10 bits of the PTEVA, concatenated together; this must be true if the pagetables are to be referenced as data using virtual addresses. Each time a reference misses in the cache, the low-order 18 bits of the RPTEVA are set with the offset of the root pagetable entry (bits 37:22 of the GVA, plus two low-order zeros). If the first-level pagetable entry (addressed with PTEVA) is not in the cache, then the RPTEVA is used to reference the root pagetable entry. As with the PTEVA, setting the top 20 bits corrupts the remaining bits.

### 4.1.4. RPTM Registers

The GSN registers provide access to four of the 256 global segments. The root pagetable map (RPTM) registers point to the root pagetable pages that map these four active segments. These registers are 20 bits wide, and should contain the physical page number of the root pagetable for their respective segments. These 20 bits are concatenated with the low-order 12 bits from the RPTEVA to form the physical address of the RPTE.

### 4.1.5. GVA Register

The global virtual address (GVA) register is another special 38-bit address register. The GVA holds the last global virtual address used to access the cache or bus. This register is changed by all processor references, so is of little use to the programmer. In essence, it is a temporary register used during address translation.

## 4.2. Global Register

The cache controller provides a single general-purpose, 38-bit register, called G. Like all other registers in control space, this register can only be accessed while in kernel mode. Because it is protected from user accesses, in can hold the virtual or physical address of a communication area in main memory, to be used during error recovery.

### 4.3. System Timers

The cache controller provides 3 timers for use by the operating system. One timer, T0, is a free-running 64-bit timer to be used for timestamps and other internal uses. The other two, T1 and T2, are 32 bit interval timers, that interrupt the processor at the end of the interval. All three timers may be started and stopped by setting a corresponding bit in the Mode register (see below). When the bit is a 1 the timer is running, and when 0 it is stopped. The interval timers clear their mode bits when their intervals expire.

#### 4.3.1. 64-bit Timer

Timer T0 is 64 bits and is incremented on every processor cycle. Assuming a minimum cycle time of 100ns, this timer will take over 50,000 years to wrap-around. While it is possible to stop this timer to obtain an accurate reading, it may be desirable to read it while running. In this case, the counter should be read twice. The values read should differ only in the least significant one or two bytes: depending on how long it takes to read these registers twice. If they differ in a more significant byte, the process should be repeated.

#### 4.3.2. Interval Timers

The other two timers, T1 and T2, are 32-bit interval timers. These timers count from –interval up to 0. When zero is reached, the mode bit is cleared, halting the timer (note that the counter may not halt exactly at 0, but will stop within a few cycles).

When the interval expires, the appropriate bit is set in the interrupt status register. If this interrupt is not masked, then the processor will handle it. The interrupt bit remains set until it is cleared by software.

### 4.4. Mode Register

The Mode Register, Mode, is used to control certain important functions of the cache controller. Bits Mode<2:0> control the performance counters, described below. Bits Mode<4>, Mode<5>, and Mode<6> control the timers T0, T1, and T2. If the bit is a one, then the timer is running, if it is zero the timer is stopped. When an interval timer expires, an interrupt is generated and the timer turns itself off (clearing the Mode bit).

**Table 4.3: Bit Assignments in Mode Register**

| Bits in Mode | Description |
|---|---|
| 0-2 | Mode for performance counters. |
| 3 | Reserved to Garth. |
| 4 | Enable timer T0. |
| 5 | Enable timer T1. |
| 6 | Enable timer T2. |
| 7 | Reserved to David. |

### 4.5. Slot Id Register

The Slot Id Register contains a read-only 4-bit value that uniquely defines the processor's slot in the SpurBus backplane. This value is hardwired on the backplane, and directly read in to the register. This register can be read normally, but although writes may be executed, they have no effect on its contents. When reading the register, its value is placed on the least significant 4 bits of the data word.

### 4.6. Exception Status Registers

There are 3 special registers that support exception processing: IStatus, IMask, and FEStatus. These registers are discussed in Section 5.

### 4.7. Performance Counters

A number of counters are provided by the cache controller to aid in the performance analysis of the working multiprocessor. These counters can be used to measure important performance metrics without perturbation to the measured system. Counters are provided both for architectural events, such as cache misses and bus operations, and calculating costs of the implementation, such as the number of cycles a processor is idle while a coherency-related operation is taking place. Many of the latter type involve the partition of the cache controller that services processor requests for the cache, which is referred to as the PCC in this document.

There are only 16 counters, and many more interesting events to measure. In order to provide a rich selection of performance metrics, five counter modes were implemented, each mode counting a different subset of events (see Table 4.4). One mode shuts off the counters so that they may be initialized and read. Three modes measure the frequency of different types of cache accesses; the three modes, Perf_User, Perf_Kernel, and Perf_Both, distinguish between user and kernel mode accesses. The final mode, Perf_Snoop, counts coherency-related events (see Section 6 for a description of the protocol).

The counters are listed in Table 4.5, along with the events measured while in one of the cache modes, or in the snoop mode. Only one event is counted in all modes.

Table 4.6 lists all the events that are measured. Note that some of the counters measure events that may not be the desired, final metric; in these cases some postprocessing will be required to obtain a more useful metric. Formulas for many of the final metrics are given in

**Table 4.4: Performance Counter Modes**

| Mode | Mode<2> | Mode<1> | Mode<0> | Description |
|---|---|---|---|---|
| Perf_Off | 0 | 0 | 0 | Do not measure any events |
| Perf_Snoop | 0 | 0 | 1 | Measure statistics for coherency protocol |
| Perf_User | 1 | 0 | 1 | Measure cache statistics for user programs |
| Perf_Sys | 1 | 1 | 0 | Measure cache statistics for the kernel |
| Perf_Both | 1 | 1 | 1 | Measure cache statistics for user and kernel |

| Table 4.5: Counters by Mode | | |
|---|---|---|
| Counter Number | Counter Name in Mode: Perf_User, Perf_Sys, Perf_Both | Counter Name in Mode: Perf_Snoop |
| C0 | Prefetches | BusWait |
| C1 | FetchesSuperset | MasterWFI |
| C2 | ReadsSuperset | MasterRS |
| C3 | WritesSuperset | MasterRFO |
| C4 | PrefetchHits | WaitWFI |
| C5 | Fetches | WaitUpdate |
| C6 | Reads | WFIInv |
| C7 | Writes | RFOInv |
| C8 | RPTEsSuperset | OwnRS |
| C9 | UPTEMisses | OwnDirty |
| C10 | Writebacks | OwnPrivateRS |
| C11 | DirtyMisses | OwnRFO |
| C12 | BusWait | PCCBusyCache |
| C13 | MasterWFI | PCCBusyState |
| C14 | External1 | WFIInterference |
| C15 | External2 | WBInterference |

Table 4.7.[4]

Each counter is 32-bits, and will therefore wrap-around in no less than 4 minutes (with a 100ns clock). The long interval between samples reduces the measurement distortion to a negligible amount. Since the counters may be turned off, by setting the mode to Perf_Off, it is not necessary to read them while running. Rate information can be obtained by reading the counters in conjunction with sampling the 64-bit timer, T0.

Most interesting events can be measured within the cache controller. In addition, external events can be counted in one of the cache modes with two cache controller chip pins, EXT_STAT<0:1>, which are sampled on Phi3. Usually, these pins will be used to count instruction issue and instruction buffer hit rates, although any other event with the correct timing constraints can also be measured.

Complete information on coherency protocol performance requires measuring additional events on the backplane, using an external hardware monitor. Examples of metrics in this category are the number of times a snoop could have responded with data, but didn't, because of the semantics of the Berkeley Ownership protocol; or, the particular bus operations that are performed on certain addresses and the frequency with which they occur.

[4] Cf. [Egge88] for additional metrics.

**Table 4.6: Description of Events Counted**

| Counter Name | Cache Modes | Snoop Mode | Description |
|---|---|---|---|
| BusWait | X | X | Number of cycles the PCC waited for a bus transfer to complete, including arbitration. |
| DirtyMisses | X | | Number of extra misses to check the PTE page dirty bit on a write. |
| External1 | X | | Number of events on EXT_STAT<0>. |
| External2 | X | | Number of events on EXT_STAT<1>. |
| Fetches | X | | Number of instruction fetches. |
| FetchesSuperset | X | | Number of instruction fetches + instruction fetch misses. |
| MasterRFO | | X | Number of ReadOwn bus operations generated. |
| MasterRS | | X | Number of Read bus operations generated. |
| MasterWFI | X | X | Number of WriteInv bus operations generated, including those changed to Read by an external invalidation. |
| OwnDirty | | X | Number of times this cache transferred an owned dirty block transfer. |
| OwnPrivateRS | | X | Number of times this cache was a private owner of a block on an external Read. |
| OwnRFO | | X | Number of times this cache was owner of a block on an external ReadOwn. |
| OwnRS | | X | Number of times this cache was owner of a block on an external Read. |
| PCCBusyCache | | X | Number of cycles the PCC ignored a snoop request for the cache (plus a state update) & then modified its state. Includes the PCC's receipt of the snoop's request signal. |
| PCCBusyState | | X | Number of cycles the PCC ignored a snoop request to update its state (no data transfer) & did the state update. Includes the PCC's receipt of the snoop's request signal. |
| PreFetches | X | | Number of instruction prefetches. |
| PreFetchHits | X | | Number of instruction prefetch hits. |
| Reads | X | | Number of data reads. |
| ReadsSuperset | X | | Number of data reads + data read misses. |
| RFOInv | | X | Number of invalidations (no data response) due to an external ReadOwn. |
| RPTEsSuperset | X | | Number of RPTE references + RPTE misses. |
| UPTEMisses | X | | Number of UPTE misses (or RPTE references). |
| WaitWFI | | X | Number of cycles the PCC waited for WriteInv to complete (including arbitration). This includes the WriteInv's that are canceled by an external invalidation of the cache block. It does not include any interruptions by the snoop to update the PCC state. |
| WaitUpdate | | X | Number of cycles the PCC was idle while the snoop updated its state on a cache flush (of a clean block only). |
| WBInterference | | X | Number of WriteInvs canceled by an external ReadOwn or WriteInv. |
| WFIInterference | | X | Number of WriteInvs canceled (& changed to ReadOwn) by an external ReadOwn. |
| WFIInv | | X | Number of invalidations due to an external WriteInv. |
| Writebacks | X | | Number of blocks written back to memory (includes flushes & copybacks that are canceled by an external invalidation). |
| Writes | X | | Number of data writes. |
| WritesSuperset | X | | Number of data writes + data write misses + DirtyMisses + MasterWFIs. |

**Table 4.7: Formulas for Typical Memory System Metrics**

| Metric | Mode | Formula | Description |
|---|---|---|---|
| FetchHits | Cache | C5 - (C1 - C5) | Eliminate fetch misses from total fetches. |
| ReadHits | Cache | C6 - (C2 - C6) | Eliminate read misses from total reads. |
| WriteHits | Cache | C7 - (C3 - C7 - C11 - C13) | Eliminate write misses from total writes. |
| UPTERefs | Cache | (C5 + C6 + C7) - (FetchHits + ReadHits + WriteHits) | Eliminate hits from total cache accesses. |
| UPTEHits | Cache | UPTEs - C9 | Eliminate UPTE misses from total UPTE accesses. |
| RPTEHits | Cache | C9 - (C8 - C9) | Eliminate RPTE misses from UPTE misses. |
| TotalBusWait | Cache | C12 + (C1 - C5) + (C2 - C6) + (C3 - C7) + C9 + (C8 - C9) + C10 | The total number of bus cycles: BusWait + an additional cycle for each bus operation. This is a slight overestimate because Writebacks includes canceled copybacks and flushes. |
| BusContention | Cache | TotalBusWait - ((C1 - C5) + (C2 - C6) + (C3 - C7 - C13) + C9 + (C8 - C9) * 18) - (C10 * 16) - (C13 * 15) | The number of additional cycles that memory accesses take because of contention for the bus: TotalBusWait - each type of operation * its nominal bus cycle cost. This is a slight overestimate because Writebacks includes canceled copybacks and flushes. |
| CacheDelay | Snoop | C12 - (C11 + C8) * 3 | Eliminate from PCCBusyCache the number of times the PCC was owner on an external request * the number of cycles for the PCC to update its state. CacheDelay includes the cycle in which the PCC receives the snoop's request. |
| CacheTransfer | Snoop | 1 + C12 + 2 + 10 | The time to carry out cache-to-cache transfers: the number of cycles for the snoop's request to the PCC for the cache + PCCBusyCache + the PCC's handling of the request other than for state update + the snoops actual use of the cache (including cleanup after the bus transfer). |
| StateDelay | Snoop | C13 - (C6 + C7) * 3 | Eliminate from PCCBusyState the number of times the PCC updated its state for the snoop (no data transfer) * the number of cycles/update. This assumes that the PCC was in the midst of a cache hit (a miss would have added another cycle to the delay) & that the cycle in which the PCC receives the snoop's request is included in the delay. |
| TotalWaitWFI | Snoop | C4 + C1 | An extra cycle/WriteInv must be added to get the total PCC idle cycles during WriteInvs. |
| WaitUPdateOPs | Snoop | C5 / 5 | Number of cache flushes to clean blocks. The divisor includes the snoop acknowledgement cycle. |
| TotalWaitUpdate | Snoop | C5 + WaitUPdateOPs | An extra cycle/state update must be added to get the total PCC idle cycles on a cache flush of clean blocks. |
| TotalBusWaitSnoop | Snoop | C0 + MasterRS + MasterRFO + MasterWFI | An extra cycle/bus operation must be added to get the total PCC idle cycles during bus transfers. This is a slight underestimate, because copybacks are not counted. |

## 5. Errors, Faults, Interrupts, and System Reset

The memory system generates several levels of exceptions, each with different semantics. *Interrupts* are generated by external events, and are a result of normal operating behavior. *Faults* occur when a virtual memory access cannot be completed without software intervention. A typical fault is caused by referencing an invalid page (i.e., page fault). *Errors* occur when some physical exception occurs in the memory system; most errors result from hardware problems (e.g., parity error) but some can be caused by software problems (e.g., corrupt pagetable entries). *System Reset* forces the system into a reset state, and causes it to begin a bootstrap sequence.

### 5.1. Interrupts

Interrupts fall into two classes: internal and external interrupts. Internal interrupts are generated by devices local to the processor board, such as the UART and interval timers. External interrupts are generated off of the processor board, and are memory mapped into the physical address space.

When an interrupt occurs the appropriate bit is set in the interrupt status register (IStatus). There are 32 bits in the IStatus, although currently only 19 are defined. A second register, the interrupt mask register (IMask), is used to mask out individual interrupts. Setting IMask<i> to 0 disables the *ith* interrupt from interrupting the processor. An interrupt is pending only if there is an unmasked interrupt posted:

Interrupt = IMask<0>&IStatus<0> | IMask<1>&IStatus<1> | ... | IMask<31>&IStatus<31>

All interrupts are masked by both the K_AllTraps and K_Interrupt bits in the kernel processor status word (KPSW). If the interrupt is not masked, the processor completes the outstanding memory request, then clears K_AllTraps and begins execution at address 0x1040 (the interrupt trap vector). All memory system exceptions (faults and errors, as well as interrupts) cause the processor to switch into kernel mode, copying K_CurMode to K_PrevMode and clearing K_CurMode.

Both IStatus and IMask are read and modified using the usual byte-wise load and store operations, however the store operation has slightly different semantics for the IStatus register. To prevent interrupts from being lost, the new contents of IStatus becomes the bitwise AND of the old contents and the 1's complement of the byte being "written". This allows the programmer to selectively clear individual bits (by putting a '1' in the appropriate bit position) after servicing the interrupt, without unintentionally affecting other bits in the register. The addresses of the registers are listed in Table 4.2, and the assignment of bits within the register is listed in Table 5.1.

External interrupts are generated by writing a single word to particular addresses in the physical address space. The interrupts are mapped into slot-space for each processor board. The address to generate interrupt i for slot j is:

$$0xF<j>0XXX<xx i_3 i_2><i_1 i_0 00>$$

where j is a 4 bit field. If a processor is in slot j, then the write will complete normally. However, if there is no board in slot j, then the write will end in a bus-timeout error (see below). Note

| Table 5.1: Interrupt Status/Mask Register Assignments | |
|---|---|
| Bit in IStatus/IMask | Description of Interrupt |
| 0 | External Interrupt 0 |
| 1 | External Interrupt 1 |
| 2 | External Interrupt 2 |
| 3 | External Interrupt 3 |
| 4 | External Interrupt 4 |
| 5 | External Interrupt 5 |
| 6 | External Interrupt 6 |
| 7 | External Interrupt 7 |
| 8 | External Interrupt 8 |
| 9 | External Interrupt 9 |
| 10 | External Interrupt 10 |
| 11 | External Interrupt 11 |
| 12 | External Interrupt 12 |
| 13 | External Interrupt 13 |
| 14 | External Interrupt 14 |
| 15 | External Interrupt 15 |
| 16 | Completion of interval timer T1 |
| 17 | Completion of interval timer T2 |
| 18 | UART interrupt |
| 19 | Auxilary Interrupt |

that if some other board (e.g., memory) is in slot j, the write may complete with unintended results.

## 5.2. Faults

Faults are generated when a processor reference cannot be completed without software intervention. The current instruction is aborted, and control transferred to address 0x1040. The CPU clears the K_AllTraps bit in the KPSW to mask out the fault while it is being handled. The fault handler will normally clear the K_Faults bit and re-enable K_AllTraps very quickly, because it is dangerous to run with traps disabled. As long as either K_AllTraps or K_Faults is 0, the CPU will ignore faults. If a fault occurs and is ignored, the cache controller will generate an error in addition to the new fault. Since errors are not masked by K_AllTraps, the system has a chance to recover or at least debug this condition. However errors can result in lost process state; therefore they should be strongly avoided.

Faults are similar to interrupts in that they cause a corresponding bit to be set in the Fault/Error status register (FEStatus); FEStatus is used to post errors, as well as faults. Faults differ from interrupts because they cannot be masked individually. All of the faults are listed in Table 5.2, along with their bit position in the register.

| Table 5.2: Fault Bits in the FEStatus Register | |
|---|---|
| Bit in FEStatus | Description of Fault |
| 16 | Protection Violation |
| 17 | Page Fault (Reference to page w/o PTEPageValid set) |
| 18 | Reference Fault (Reference to page w/o PTEPageReferenced set) |
| 19 | Dirty Bit Fault (Write to page w/o PTEPageDirty set) |
| 20 | Bad SubOp Fault (Invalid SubOp in Ld_External or St_External) |
| 21 | Try Again (Bus reference generated a try-again status) |
| 22 | Illegal Cache Operation (on non-cacheable page). |

## 5.3. Errors

Errors are generated by hardware problems or serious operating system failures. Memory parity errors and bus protocol violations are examples of hardware failures that result in errors. Software problems, such as corrupt pagetable entries, can cause bus errors or bus timeouts to occur.

Errors are handled similar to faults. The error condition is posted by setting the appropriate bit in the FEStatus register (see Table 5.3). When an error is detected, the CPU switches to physical mode and begins execution at address 0x1010 in EPROM. The K_Error and K_AllTraps bits are cleared, causing all further exceptions to be ignored. *The error handler must be EXTREMELY CAREFUL not to generate a second error until it has re-enabled K_Error; if the error handler causes a second error, then the processor will hang until a system reset.*

Errors can occur "out-of-band", as well as in response to virtual and physical memory references. An out-of-band error is caused by a SpurBus reset. These resets result from a number of error conditions on the bus. Some bus errors result in cache state becoming inconsistent (the snoop copy differs from the processor copy). In these cases, the cache controller that is master of the transaction generates a SpurBus reset, and reports the error condition that resulted in the inconsistent state. All other processors will see a simple bus reset. The error handler must patch up the cache state before resuming normal operation.

It is important to note that an error may cause the loss of a register window, if the error occurs while in the window overflow handler. Thus some errors may cause processes to be killed, and may even cause a system reset, if the lost window is within the kernel.

### 5.3.1. Recovering from Inconsistent Cache Errors

When an error results in an inconsistent cache, it is necessary to restore consistency before switching back to virtual addressing mode.

The inconsistency arises because the snoop copy of the cache state is updated *before* the bus operation begins, while the processor copy is not updated until *after* successful completion. Thus in many cases an inconsistency can be detected just by locating a block whose state and tags differ between the two copies (there should be exactly one). Consistency can be restored by applying the following rule: if the block is marked dirty, copy the processor copy to the snoop copy, if it is clean, set both copies of the state to Invalid. The first case occurs when a write-back

| Table 5.3: Error Bits in the FEStatus Register | |
|---|---|
| Bit in FEStatus | Description of Error |
| 0†† | System Reset (A long reset occurred on the SpurBus) |
| 1 | Bus Reset (A reset was detected on the SpurBus) |
| 2† | A Try-Again bus status occurred, resulting in inconsistent cache state |
| 3† | A bus timeout occurred, resulting in inconsistent cache state |
| 4† | A bus error occurred, resulting in inconsistent cache state |
| 6 | A bus timeout occurred, however the cache remains consistent |
| 7 | A bus error occurred, however the cache remains consistent |
| 8 | Internal error: SBC inconsistency. |
| 9 | Internal error: bad sbc request code. |
| 10 | Internal error: bad sbc ack code. |
| 11 | Fault occurred while CPU was ignoring faults. |
| † error generates SpurBus reset<br>†† Does not affect the ERROR signal to the CPU. | |

to memory was in progress. The second case occurs when some other operation was being performed. In some special cases (for example, while processing a dirty bit miss), there may not be any blocks that are marked inconsistent. In these situations, flushing the cache will eliminate the inconsistency. This works because dirty blocks are known to be OK and can be written back to main memory; the inconsistent block must be clean, and therefore will be invalidated.

### 5.3.2. Atomic Operations and Error

The **test_and_set** instruction provides a simple read-and-set atomic primitive to aid in synchronization. Unfortunately, in the implementation of the SPUR prototype, errors can cause the "atomic" operation not to be atomic; i.e., the "set" portion of the "read and set" can be lost. This type of error will generally result in a system reboot. It is probably possible to avoid this problem in software by following a restricted lock/unlock convention, with support in either the error handler or in-line code to detect the problem and re-execute the **test_and_set**.

# 6. Cache Coherency Protocol

## 6.1. Functional Overview

The Berkeley Ownership coherency protocol is based on the concept of processors owning cache blocks. The processor that owns a block may update it without initiating a bus transfer. If a cache has a block but does not own it, the processor must first obtain ownership before performing the update. Ownership is obtained by a broadcast to other caches, causing them to invalidate their copies of the block.

The *snoop* portion of the cache controller monitors the system bus for operations affecting blocks contained in its cache. The snoop compares the virtual addresses of all bus transactions with its own copy of the cache's virtual address tags. When an address match is made, the snoop performs consistency-preserving operations, based on the type of bus request and the state of the cache block.

## 6.2. The Coherency States

Each cache block has associated with it a coherency state that (1) denotes the coherency status of the block in the entire multiprocessor system, (2) restricts what operations are allowed on that block and (3) defines what coherency-preserving responsibilities the cache controller has in handling it. The states, their values, and their precise semantics are defined in Table 6.1.

## 6.3. The Coherency Protocol Bus Operations

A cache block's state can only change when a bus operation occurs, either originated by the local or an external processor. The type of bus operation, its origin, and the prior state determine the new state. The SPURBus supports conventional read and write operations, as well as two additional cache coherency protocol specific operations. A complete description of the bus operations (from the point of view of the initiating processor) is given in Table 6.2).

**Table 6.1: Cache State Semantics**

| State Name | Value | Semantics |
|---|---|---|
| Invalid | 0 | The block is invalid. |
| UnOwned | 1 | The block is valid and possibly shared in other caches. The entry is readable, but cannot be written locally without first acquiring ownership. |
| OwnPrivate | 3 | The block is the only copy in the system outside memory; therefore it can be written locally. The cache controller must provide the data to a requesting cache and flush the block to memory on replacement. |
| OwnShared | 2 | The block cannot be updated without broadcasting an invalidation signal. The cache controller must provide the data to a requesting cache and flush the block to memory on replacement. |

**Table 6.2: Bus Operations**

| Bus Operation | Value | Description |
|---|---|---|
| Read | 0 | A conventional read of a cache block. The state value of the block becomes UnOwned. |
| ReadOwn | 2 | Similar to Read, except that other cache controllers invalidate cache blocks with matching tag values. The state value of the block becomes OwnPrivate. |
| WriteInv | 5 | A one-word dummy write. Causes the other cache controllers to invalidate cache blocks with matching tag values. The state value of the block becomes OwnPrivate. |
| Write | 3 | A cache block is updated to memory, but no cached copies are invalidated. Used for copybacks on block replacement. The state value of the block becomes Invalid. |

## 6.4. The Protocol from the Point of View of the Processor

When a processor load hits in the cache, the appropriate word is provided to it. (See Figure 6.1 for the state transition diagram of the Berkeley Ownership protocol. In this figure the boxes are the states, the solid lines are bus operations that cause processor-initiated state updates, and

State Transitions in Berkeley Ownership

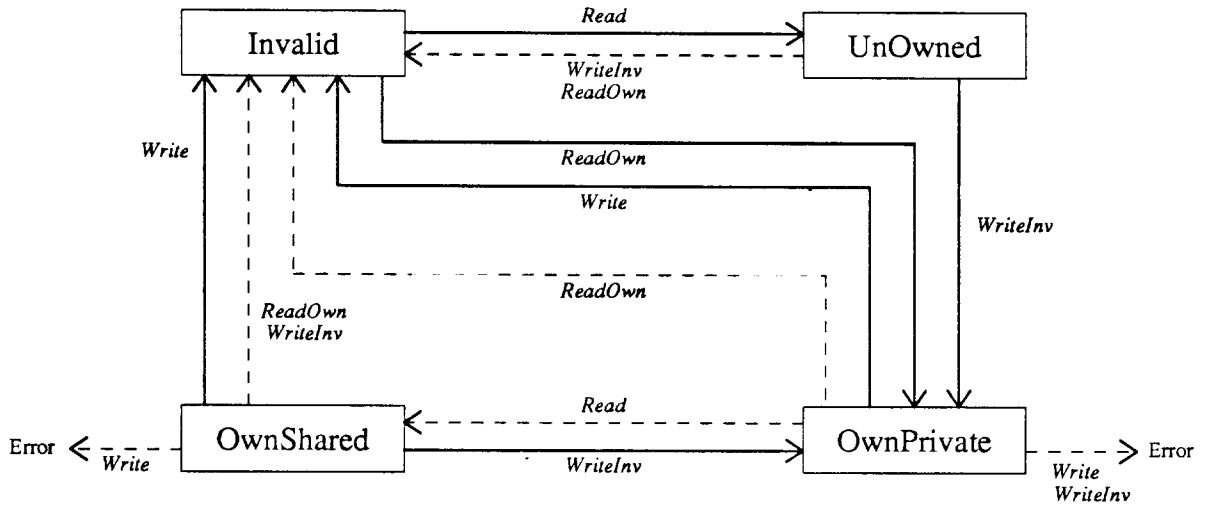


Figure 6.1: Transition Diagram of the Berkeley Ownership Protocol

the dashed lines are snooped bus transactions that resulted in snoop-initiated changes.) On a miss, if the current cache block is dirty, it must first be flushed back to memory with Write. If the PTEPageCoherency bit is set or if the instruction is a *ReadOwnership* instruction (e.g., **ld_32_ro**), the processor obtains ownership by using ReadOwn; otherwise it issues Read. In the former case the state becomes OwnPrivate, and in the latter UnOwned.

If a processor store hits on a block with a state of OwnPrivate, the update can be made without issuing a bus operation. If the state of the block is OwnShared or UnOwned (and the tags match), then the processor signals (with WriteInv) its intention to write before it modifies the block. The snoops of the other processors invalidate matching blocks in their caches. If the state is Invalid or if the tags do not match, the cache controller reads the requested block with ReadOwn. On cache misses, a dirty block is replaced with Write before the requested block is read. For all writes the final state of the block is updated to OwnPrivate.

### 6.5. The Protocol from the Point of View of the Snoop

The snoop accesses the cache only to invalidate a block written by another processor or to provide an owned block for an external read request. For all bus operations no action is taken when the block is not in the cache. If the bus operation is Read or ReadOwn and the block's state is OwnPrivate or OwnShared, the snoop obtains use of the cache and provides the data in an inter-cache transfer. The snoop changes the state to OwnShared if the bus operation is Read, and to Invalid if ReadOwn. If the cache state is UnOwned and the ReadOwn operation occurs, the snoop invalidates its copy only.

WriteInv occurs when a processor with a UnOwned or OwnShared copy wants to update it. If the snoop's copy of the state is UnOwned or OwnShared, the snoop invalidates its copy. If the state is OwnPrivate, the snoop generates an error, since this is semantically illegal, and should not occur.

### 6.6. Atomic Operations

Operating systems and parallel programs frequently use spin locks to serialize access to shared data and critical sections. To minimize coherency overhead, we suggest that spin locks be implemented by a test-and-test-and-set algorithm [Sega84] (see Figure 6.2).

```
/* Lock via the test-and-test-and-set algorithm. */
    spin(lockp)
    int *lockp;
    {
            while (*lockp) {
                    while (*lockp); /* spin using ld_32 */
                    while (! *lockp) {
                            if (! test_and_set(*lockp)) {
                                    /* test_and_set can fail due to errors. */
                                    if (*lockp) {
                                            return; /* Got the lock */
                                    } else {
                                            /* test_and_set failed, see Section 5. */;
                                    }
                            } else {
                                    break; /* Didn't get the lock. */
                            }
                    }
            }
    }
```

Figure 6.2: Locking Sequence

The test-and-test-and-set algorithm minimizes bus traffic in the case of contention for a lock. In this situation, all processors other than the owner of the lock repeatedly check the lock to see if it has been released. The initial read brings an UnOwned copy of the block into the cache; the processor continues to read this copy with no further bus operations until the lock is released. When the lock is released, the spinning processors first read the block back into their caches, then (after determining that it is free) execute the **test_and_set** instruction. The first processor to execute **test_and_set** will issue a WriteInv to obtain ownership. The losing processors will have the block "stolen" out from under them by the WriteInv, and have to issue a ReadOwn operation to get it back.

If the initial read is omitted, every **test_and_set** instruction by the spinning processors will cause a bus operation (ReadOwn). The cost of the test-and-test-and-set algorithm is one additional bus operation each time a lock is obtained, if there is no contention. Since most locks have little contention, it may be better to try a single **test_and_set** first, then fall back to the spin algorithm only when there is contention.

## 6.7. Coherency Optimization

As described in Section 2, the cache controller checks the PTEPageCoherency bit on cache misses. If it is set, ownership is obtained using the ReadOwn operation, regardless of the processors request. The processor can thereafter write to the block without causing additional bus operations. If PTEPageCoherency is not set, two bus operations will be necessary when a block is read before it is written; the initial read will generate a Read operation, and the subsequent

write will cause a WriteInv. The PTEPageCoherency bit should generally be set for private data pages, e.g., the stack and private heap, since these will not usually be shared by multiple processors. The bit should generally not be set for shared data pages and code pages.

The *ReadOwnership* instructions, e.g., **ld_32_ro**, cause ownership to be obtained on a cache miss, regardless of the PTEPageCoherency bit. These instructions can obtain the same bus traffic optimizations even when the page is normally shared. An intelligent compiler or hand-coded inner loops can use these variants of the load instructions to improve performance.

## 7. Virtual Cache Algorithms

The physical address caches in most commercial computers are completely transparent to the operating system. In other words, the operating system executes correctly whether the cache exists or not, and only the performance is affected. This is not the case for virtual address caches, such as the one used in SPUR.

An operating system running on the SPUR hardware must enforce certain restrictions and take appropriate actions to maintain intra-cache and cache-memory consistency. These actions are required even in a single processor system because the problems are due to the virtual address cache, not the multiple processors. This section summarizes these responsibilities.

### 7.1. Restricting Virtual Address Synonyms

A *virtual address synonym*, also known as an *alias*, exists when two (global) virtual addresses map to the same physical address. When a cache is indexed using the virtual address then it is possible for the same datum to reside in two separate cache lines. If one copy is modified, the other copy becomes out of date. Clearly, inconsistencies of this kind can result in erroneous results.

This problem exists for both virtual address caches and some physical address caches (where addresses are translated in parallel with the cache access). High-performance systems typically throw hardware at the problem, increasing the cost of the processor node. Expensive hardware solutions are undesirable for a multiprocessor workstation, where low unit cost is important. Instead, SPUR uses a (primarily) software solution to eliminate synonyms, and therefore prevent the problems from arising.

The operating system is responsible for preventing synonyms. This means the operating system *must* maintain a one-to-one mapping between the global virtual address space and the physical address space. This restriction guarantees that every memory block always maps to the same cache line each time it is brought into the cache.

This restriction can be relaxed for pages that are read-only since synonyms are only a problem when they are written. Thus, it is possible to implement copy-on-write since pages are made read-only until they are copied. Because protection is maintained on the *virtual* page, all virtual pages involved in a synonym must be read-only.

### 7.2. I/O Considerations

The Berkeley Ownership protocol prevents cache coherency problems between processors. However, the NuBus I/O devices do not participate in the cache coherency protocol. Therefore, I/O buffers and virtual memory pages must be flushed from all the caches before I/O is initiated. If the caches are not flushed, then stale data may be written to secondary storage, or newly read data may be overwritten by a dirty cache block. The virtual memory page, or page containing the I/O buffer, should also be marked invalid or non-cacheable to prevent a processor from bringing a block into its cache.

Flushing a single page from all the caches can be done in two ways. The first is to interrupt all the other processors and request that they flush the page from their caches. An interrupt level could be dedicated to this function; the interrupt handler would read the virtual page number from a communication region in the system segment. As each processor finishes, it updates a table in the communication region. The initiating processor waits until all other processors have

completed before resuming execution.

The second way to flush a page is to take advantage of the cache coherency protocol and purge the other caches by obtaining an exclusive copy of each block in the page. The blocks can then be flushed back to memory using the FLUSH operation.

Pseudo-code for this algorithm is presented in Figure 7.1. The page must be marked invalid (by clearing PTEPageValid) to prevent other processors from accessing the page during the flush. The blocks from the page can be brought into the local processor's cache using the **ld_32_ri** instruction; this instruction ignores the valid bit in the PTE, and obtains an exclusive copy of the cache block. The block can then be flushed back to memory using the **FLUSH** operation (see Section 5). The process should not be rescheduled to another processor while this code is being executed.

```
#define BLOCKSIZE    32
#define PAGESIZE     4096

/* page_addr contains virtual address of page */
VmClearValidBit(page_addr);          /* Mark page invalid */
for (p = page_addr; p < page_addr+PAGESIZE; p = p + BLOCKSIZE) {
        ReadAnyways(p);              /* Fetch block with ld_32_ri */
        FlushBlock(p);               /* Flush from cache */
}
```

Figure 7.1: Selective Cache Flush Pseudo-Code

## 7.3. Modifying Pagetable Entries

Operating system writers are familiar with taking special actions when changing pagetable entries. In most systems, a translation buffer caches copies of the PTEs to improve performance; the operating system must invalidate (at least) one entry to guarantee the pagetable update is seen immediately.

In a virtual address cache, like SPUR's, some of the PTE information is cached with each block. Thus to propagate a pagetable update, each block from the affected page must be flushed from the caches. This operation is fairly expensive, so the operating system should try to reduce its frequency.

In the remainder of this subsection, we discuss changing each of the pagetable entry fields in turn. A *selective cache flush* refers to flushing all the blocks from a particular page from all the caches.

### 7.3.1. PTEPagePhysicalAddress and PTEPageValid

Clearing the PTEPageValid bit prevents a processor from bringing a block from that page into its cache (except as noted below). Processors can still access blocks that are already in their cache. A selective cache flush is required to prevent further access to the page.

The **ld_32_ri** instruction ignores the PTEPageValid bit. This instruction is useful because it allows the kernel to "steal" blocks from other caches during a selective cache flush. On the other hand, it is extremely dangerous. If the page is truly invalid, i.e., PTEPagePhysicalAddress is garbage, executing **ld_32_ri** can cause a bus error or bus timeout error. Since errors can cause system failures, the operating system must be very careful when using this instruction.

Before changing the mapping of a virtual page (i.e., changing PTEPagePhysicalAddress), the page should first be flushed from the caches. This prevents a processor from accidentally reading a block from the old physical page that has remained in the cache.

Similarly, before allocating a physical page, all blocks from the previous mapping should be flushed from the caches. Otherwise main memory could be overwritten by old data, because each cache block contains its corresponding physical address. A block replacement or cache flush could overwrite memory despite the change to the mapping (in the PTE).

### 7.3.2. PTEPageProtection

When a block is brought into the cache, the PTEPageProtection field is copied into the cache line. It is this cached copy that is checked by the hardware, not the PTE copy. Thus changes to PTEPageProtection do not affect previously cached blocks; a selective cache flush is required to propagate the update.

Sometimes the cache flush can be deferred, or even eliminated. For example, when the protection is relaxed, as from read-only to read-write, deferring the flush may result in some false protection violations, but the trap handler can easily detect these cases and resume normal execution (after flushing the offending cache block to refresh its protection field). If these false traps are infrequent, then this approach will have less overhead.

### 7.3.3. PTEPageCacheable

Switching a page from non-cacheable to cacheable does not requires any action, since no blocks from the page should be in the caches. Clearing PTEPageCacheable, on the other hand, only prevents new blocks from being brought into the caches; a selective cache flush is necessary to remove blocks that are already cached. To guarantee a consistent view of the page, the cache flush and marking the page non-cacheable should be an atomic action. This is possible by marking the page invalid during the cache flush, then simultaneously marking the page valid and non-cacheable.

### 7.3.4. PTEPageReferenced

As discussed in Section 2.3.2, PTEPageReferenced is provided to approximate a "least-recently-used" replacement policy. The hardware checks this bit only when a block is brought into the cache. To obtain "true" reference bit behavior, the operating system must perform a selective cache flush after clearing PTEPageReferenced; this will guarantee that the next reference to the page will cause a Reference fault.

We suggest a lower overhead alternative, called the *miss-bit approximation*. In this approximation, the operating system just clears PTEPageReferenced and does *not* flush the cache. Assuming that frequently referenced pages are more likely to have cache misses, this approximation should achieve comparable results with less overhead.

### 7.3.5. PTEPageDirty

As discussed in Section 2.3.2, PTEPageDirty indicates whether the page has been modified. The trap handler sets this bit *before* the page is actually modified. When writing the page to secondary storage, it is important to prevent a write by a different processor from occurring between the I/O transfer and clearing the PTEPageDirty bit. This is easily done by making the page invalid or read-only before flushing it out of the cache. Note that the page must be flushed to memory anyways to maintain coherency.

## 7.4. Re-using Segments

As described previously, each process has access to 4 segments at a time. Generally, all processes will share the same system segment, and many will share code and perhaps data segments. Nonetheless, there are only 256 segments in the system and they are clearly a limited resource.

Before a segment is re-allocated, the system must guarantee that no data from the previous use of the segment resides in any of the caches. In other words, we are deallocating the pages previously mapped to the segment and must flush the caches to prevent false hits. Because there may be many pages involved, we may want to perform a system-wide full cache flush, rather than selective flushes of individual pages.

A system-wide cache flush could be initiated by interrupting all processors, and having their interrupt handlers begin execution of the cache flush routine. The initiating processor waits until all flushes have completed, then marks the segment as clean. Multiple segments can be recycled simultaneously, which will help amortize the cost of the flushes.

Another algorithm is to have each processor flush ONLY blocks from the specified segment. Using the physical address space, we can read the virtual address tag for each line in the cache, and flush it only if the block is in the specified segment. Pseudo-code for this algorithm is presented in Figure 7.2. Note that because this code uses the physical address space, it can only be executed in kernel mode.

```
#define BLOCKSIZE   32
#define CACHESIZE   131072
#define SEGMASK     0xFF000000

/* flush_seg contains segment number to flush, in high-order 8 bits */
for (i = 0; i < CACHESIZE; i = i + BLOCKSIZE) {
        if (ReadVirtualTag(i) & SEGMASK == flush_seg) {
                FlushBlock(i);          /* Flush from cache */
        }
}
```

Figure 7.2: Segment Cache Flush Pseudo-Code

## 8. References


[Egge88] Eggers, S. J., "Simulation Analysis of Data Sharing Support in Shared Memory Multiprocessors", Ph.D. thesis, in progress, University of California, Berkeley, completion, 1988.

[Gibs85] Gibson, G., "SPURBUS Specification", *Proc. of CS292i: Implementation of VLSI Systems*, R. H. Katz (editor), University of California, Berkeley, September 1985.

[Hill86] Hill, M. D., 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. Hodges, R. H. Katz, J. Ousterhout and D. A. Patterson, "SPUR: A VLSI Multiprocessor Workstation", *Computer*, Vol. 19, No. 11 , November 1986, pp. 8-22.

[Kate83] Katevenis, M. G. H., "Reduced Instruction Set Computer Architectures for VLSI", UC Berkeley, Technical Report No. UCB/Computer Science Dpt. 83/141, October 1983. Ph.D. Dissertation.

[Katz85] Katz, R. H., S. J. Eggers, D. A. Wood, C. L. Perkins and R. G. Sheldon, "Implementing a Cache Consistency Protocol", *Proc. 12th International Symposium on Computer Architecture*, Boston, Mass. , pp. 276-283, June 1985.

[Sega84] Segall, Z. and L. Rudolph, "Dynamic Decentralized Cache Schemes for an MIMD Parallel Processor", *Proceedings of the 11th International Symposium on Computer Architecture*, Vol. 12, No. 3 , June 1984, pp. 340-347.

[Sign83] Signetics, Signetics DUART SCN68681 (January 1983).

[Tayl88] Taylor, G. S., "An Analysis of a RISC for Lisp", Ph.D. Dissertation, University of California, Berkeley, in preparation, 1988.

[Texa83] Texas Instruments, NuMachine NuBus Specification, part number TI-2242825-0001 (1983).

[Wood86] Wood, D. A., 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", *Proc. Thirteenth International Symposium on Computer Architecture*, Tokyo, Japan , pp. 358-365, June 1986.