Virtual Memory in Contemporary Microprocessors

THIS SURVEY OF SIX COMMERCIAL MEMORY-MANAGEMENT DESIGNS DESCRIBES HOW EACH PROCESSOR ARCHITECTURE SUPPORTS THE COMMON FEATURES OF VIRTUAL MEMORY: ADDRESS SPACE PROTECTION, SHARED MEMORY, AND LARGE ADDRESS SPACES.

Bruce Jacob
University of Maryland

Trevor Mudge
University of Michigan

Virtual memory is a technique for managing the resource of physical memory. It gives an application the illusion of a very large amount of memory, typically much larger than what is actually available. It protects the code and data of user-level applications from the actions of other programs but also allows programs to share portions of their address spaces if desired. It supports the execution of processes partially resident in memory. Only the most recently used portions of a process's address space actually occupy physical memory—the rest of the address space is stored on disk until needed. For a primer on virtual memory, see our companion article in *Computer* magazine.[1]

Most contemporary general-purpose processors support virtual memory through a hardware memory management unit (MMU) that translates virtual addresses to physical addresses. Unfortunately, the various microarchitectures define the virtual-memory interface differently, and, as explained in the next section, this is becoming a significant problem.

Here, we consider the memory management designs of a sampling of six recent processors, focusing primarily on their architectural differences, and hint at optimizations that someone designing or porting system software might want to consider. We selected examples from the most popular commercial microarchitectures: the MIPS R10000, Alpha 21164, PowerPC 604, PA-8000, UltraSPARC-I, and Pentium II. Table 1 points out a few of their similarities by comparing their support for some core virtual-memory functions.

Memory management

The classic MMU, as in the DEC VAX, GE 645, and Intel Pentium architectures,[2-4] includes a translation look-aside buffer that translates addresses and a finite-state machine that walks the page table. The TLB is an on-chip memory structure that caches only page table entries (PTEs). If the necessary translation information is in the TLB, the system can translate a virtual address to a physical address without accessing the page table. If the translation information is not found in the TLB (called a TLB miss), one must search the page table for the mapping and insert it into the TLB before processing can continue. In early designs a hardware state machine performed this activity; on a TLB miss, the state machine walked the page table, loaded the mapping, refilled the TLB, and restarted the computation.

TLBs usually have on the order of 100 entries, are often fully associative, and are typically accessed every clock cycle. They trans-

0272-1732/98/$10.00 © 1998 IEEE

Table 1. Comparison of architectural support for virtual memory in six commercial MMUs.

Item	MIPS	Alpha	PowerPC	PA-RISC	UltraSPARC	IA-32
Address space protection	ASIDs	ASIDs	Segmentation	Multiple ASIDs	ASIDs	Segmentation
Shared memory	GLOBAL bit in TLB entry	GLOBAL bit in TLB entry	Segmentation	Multiple ASIDs; segmentation	Indirect specification of ASIDs	Segmentation
Large address spaces	64-bit addressing	64-bit addressing	52-/80-bit segmented addressing	96-bit segmented addressing	64-bit addressing	None
Fine-grained protection	In TLB entry	In TLB entry	In TLB entry	In TLB entry	In TLB entry	In TLB entry; per segment
Page table support	Software-managed TLB	Software-managed TLB	Hardware-managed TLB; inverted page table	Software-managed TLB	Software-managed TLB	Hardware-managed TLB/hierarchical page table
Superpages	Variable page size set in TLB entry: 4 Kbyte to 16 Mbyte, by 4	Groupings of 8, 64, 512 pages (set in TLB entry)	Block address translation: 128 Kbytes to 256 Mbytes, by 2	Variable page size set in TLB entry:4 Kbytes to 64 Mbytes, by 4	Variable page size set in TLB entry: 8, 64, 512 Kbytes, and 4 Mbytes	Segmentation/ variable page size set in TLB entry: 4 Kbytes or 4 Mbytes

late both instruction and data stream addresses. They can constrain the chip's clock cycle as they tend to be fairly slow, and they are also power-hungry—both are a consequence of the TLB's high degree of associativity. Today's systems require both high clock speeds and low power; in response, two-way and four-way set-associative TLB designs are popular, as lower degrees of associativity have far less impact on clock speed and power consumption than fully associative designs. To provide increased translation bandwidth, designers often use split TLB designs.

The state machine is an efficient design as it disturbs the processor pipeline only slightly. During a TLB miss, the instruction pipeline effectively freezes: in contrast to taking an exception, the pipeline is not disturbed, and the reorder buffer need not be flushed. The instruction cache is not used, and the data cache is used only if the page table is located in cacheable space. At the worst, the execution of the state machine will replace a few lines in the data cache. Many designs do not even freeze the pipeline; for instance, the Intel Pentium Pro allows instructions that are independent of the faulting instruction to continue processing while the TLB miss is serviced. The primary disadvantage of the state machine is that the page table organization is effectively etched in stone; the operating system (OS) has little flexibility in tailoring a design.

In response, recent memory management designs have used a software-managed TLB, in which the OS handles TLB misses. MIPS was one of the earliest commercial architectures to offer a software-managed TLB,[5] though the Astronautics Corporation of America holds a patent for a software-managed design.[6] In a software-managed TLB miss, the hardware interrupts the OS and vectors to a software routine that walks the page table. The OS thus defines the page table organization, since hardware never directly manages the table.

The flexibility of the software-managed mechanism comes at a performance cost. The TLB miss handler that walks the page table is an OS primitive usually 10 to 100 instructions long. If the handler code is not in the instruction cache at the time of the TLB miss, the time to handle the miss can be much longer than in the hardware-walked scheme. In addition, the use of the interrupt mechanism adds a number of cycles to the cost by flushing the pipeline—possibly flushing a large number of instructions from the reorder

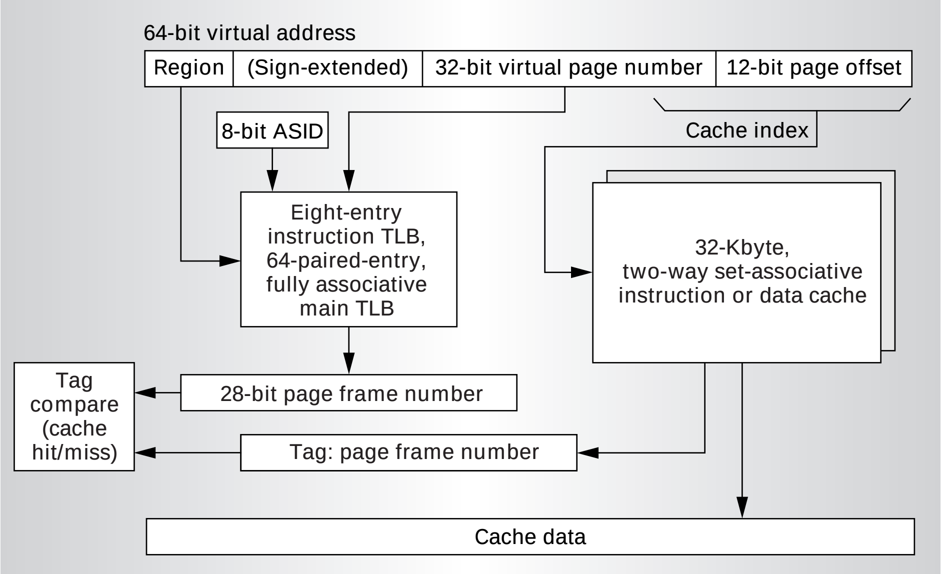
64-bit virtual address
Region
(Sign-extended)
32-bit virtual page number
12-bit page offset
8-bit ASID
Cache index
Eight-entry instruction TLB, 64-paired-entry, fully associative main TLB
32-Kbyte, two-way set-associative instruction or data cache
Tag compare (cache hit/miss)
28-bit page frame number
Tag: page frame number
Cache data

Figure 1. MIPS R10000 address translation mechanism. The split instruction and data caches are virtually indexed, both requiring an index larger than the page offset. The TLB lookup proceeds in parallel with the cache lookup. The earliest MIPS designs had physically indexed caches, and if the cache was larger than the page size, the cache was accessed in series with the TLB. ASID: address space identifier.

buffer. This can add hundreds of cycles to the overhead of walking the page table. Nonetheless, the flexibility afforded by the software-managed scheme can outweigh the potentially higher per-miss cost of the design.[7]

Given the few details presented so far, one can easily see that the use of different virtual-memory interface definitions in every microarchitecture is becoming a significant problem. More often than not, the OS running on a microprocessor was not initially designed for that processor: OSs often outlast the hardware on which they were designed and built, and the more popular OSs are ported to many different architectures. Hardware abstraction layers (for example, see Rashid et al.[8] and Custer[9]) hide hardware particulars from most of the OS, and they can prevent system designers from fully optimizing their software. These types of mismatches between OSs and microarchitectures cause significant performance problems;[10] an OS not tuned to the hardware on which it operates is unlikely to live up to its potential performance.

The following sections describe the different commercial virtual memory interfaces. First is the MIPS organization, which has the most in common with the others. Then, we concentrate on those mechanisms that are unique to each architecture.

MIPS

MIPS (Figure 1) defines one of the simplest memory management architectures among recent microprocessors. The OS handles TLB misses entirely in software: the software fills in the TLB, and the OS defines the TLB replacement policy.

Address space

The R2000/R3000 virtual address is 32 bits wide; the R10000 virtual address is 64 bits wide, though not all 64 bits are translated in the R10000. The top "region" bits divide the virtual space into areas of different behavior. The top two bits distinguish between user, supervisor, and kernel spaces (the R10000 offers three levels of execution/access privileges). Further bits divide the kernel and supervisor regions into areas of different memory behavior (that is, cached/uncached, mapped/unmapped).

In the R2000/R3000, the top bit divides the 4-Gbyte address space into user and kernel regions, and the next two bits further divide the kernel's space into cached/uncached and mapped/unmapped regions. In both architectures, virtual addresses are extended with an address space identifier (ASID) to distinguish between contexts. The 6-bit-wide ASID on the R2000/R3000 uniquely identifies 64 processes; the 8-bit-wide ASID on the R10000 uniquely identifies 256.

Since it is simpler, we describe the 32-bit address space of the R2000/R3000. User space, called kuseg, occupies the bottom 2 Gbytes of the address space. All kuseg references are mapped through the TLB and considered cacheable by the hardware unless otherwise noted in a TLB entry. The top half of the virtual address space belongs to the kernel: an address generated with the top bit set while in user mode causes an exception. Kernel space is divided into three regions: the 1-Gbyte kseg2 region is cacheable and mapped through the TLB like kuseg. The other two 512-Mbyte regions (kseg0 and kseg1) are mapped directly onto physical memory; the hardware zeroes out the top three bits to generate the physical address directly. The hardware then caches references to kseg0, but not the references to kseg1.

TLB

The MIPS TLB is a unified 64-entry, fully associative cache. The OS loads page table entries (PTEs) into the TLB, using either random replacement (the hardware chooses a TLB slot randomly) or specified placement (the OS tells the hardware which slot to choose). The TLB's 64 entries are partitioned between "wired" and "random" entries. While the R2000/R3000 has eight wired entries, the partition between wired and random entries is set by the R10000 software. The hardware provides a mechanism to choose one of the random slots. On request, it produces a random number between index values of 8 and 63, inclusive (the R10000 produces values between *N* and 63, inclusive, where *N* is set by software). This random number references only the TLB's random entries; by not returning values corresponding to wired entries, it effectively protects those entries. The TLBWR (TLB write random) instruction uses this mechanism to insert a mapping randomly into the TLB, and the TLBWI (TLB write indexed) instruction inserts mappings at any specified location. Most OSs use the wired partition to store root-level PTEs and kernel mappings in the protected slots, keeping user mappings in the random slots and using a low-cost random replacement policy to manage them.

The OS interacts with the TLB through the EntryHi and EntryLo registers, pictured in Figure 2. EntryHi contains a virtual page number and an ASID; EntryLo corresponds to a PTE and contains a page frame number and status bits. A TLB entry is equivalent to the concatenation of these structures.

The R10000 structure is similar but larger. It also has two separate EntryLo registers—one for each of two paired virtual page numbers. This allows the R10000 to effectively double the reach of the TLB without adding more entries. A single TLB entry maps every two contiguous even-odd virtual pages, though each receives its own page frame number (PFN) and status bits. The design saves die area and power. It is nearly as flexible as a 128-entry TLB but requires half the tag area—because two mappings share each virtual page number (VPN)—and half the comparators.

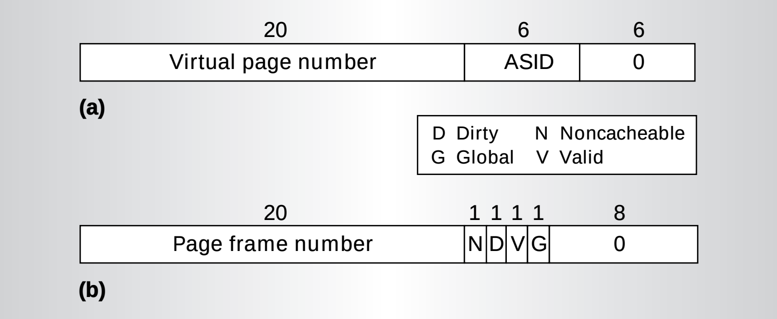

Figure 2. MIPS R2000/3000 TLB entry format: EntryHi (a) and EntryLo (b).

In the MIPS R2000/R3000, the status fields in EntryLo are

- *N, noncacheable.* If this bit is set for a TLB entry, the page it maps is not cached; the processor sends the address out to main memory without accessing the cache.
- *D, dirty.* If this bit is set, the page is writable. The bit can be set by software, so it is effectively a write-enable bit. A store to a page with the dirty bit cleared causes a protection violation.
- *V, valid.* If this bit is set, the entry contains a valid mapping.
- *G, global.* If this bit is set, the TLB ignores the ASID match requirement for a TLB hit on this page. This feature supports shared memory and allows the kernel to access its own code and data while operating on behalf of a user process without having to save or restore ASIDs.

The R10000 inherits this organization and adds more powerful control-status bits that support features such as more complex caching behavior and specification for coherency protocols. Also, if the G bit is set for either page in a paired entry, the ASID check is disabled for both pages.

When the OS reads an entry from the TLB, the hardware places the information into EntryHi and EntryLo. When the OS inserts a mapping into the TLB, it first loads the desired values into these registers. It then exe-

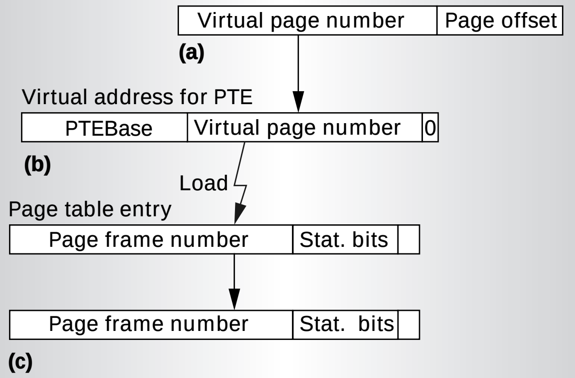

Figure 3. Using the TLB context register. The VPN of the faulting virtual address (a) is placed into the context register (b), creating the virtual address of the mapping PTE. This PTE goes into EntryLo (c).

cutes a TLBWR instruction, or it loads a slot number into the index register and executes a TLBWI instruction. Thus, the OS has the tools to implement a wide range of replacement policies.

Periodic TLB flushes are unavoidable in these MIPS processors, as there are 64 unique context identifiers in the R2000/R3000 and 256 in the R10000. Many systems have more active processes than this, requiring ASID sharing and periodic remapping. When an ASID is temporarily reassigned from one process to another, it is necessary to first flush TLB entries with that ASID. It is possible to avoid flushing the cache by flushing the TLB; since the caches are physically tagged, the new process cannot overwrite the old process's data.

Address translation and TLB-miss handling

MIPS supports a simple bottom-up hierarchical page table organization,[1] though an OS is free to choose a different page table organization. We describe the R2000/3000 translation mechanism here; the R10000 mechanism is similar. The VPN of any page in a user's address space is also an index into the user page table. On a user-level TLB miss, one can use the faulting VPN to create a virtual address for the mapping PTE. Frequently, the OS will successfully load the PTE with this address, requiring only one memory reference to handle a TLB miss. In the worst case, a PTE lookup will require an additional memory reference to look up the root-level PTE as well.

MIPS offers a hardware assist for the software TLB miss handler: the TLB context register, as depicted in Figure 3. At the time of a user-level TLB miss, the context register contains the virtual address of the PTE that maps the faulting page. The system software loads the top bits of the TLB context register, called *PTEBase*. PTEBase represents the virtual base address of the current process's user page table. When a user address misses the TLB, hardware fills in the next bits of the context register with the VPN of the faulting address. The bottom bits of the context register are defined to be zero (the R2000/3000 PTE is 4 bytes, the R10000 PTE is 8 bytes), so the faulting VPN is an index into the linear user page table structure and identifies the user PTE that maps the faulting address.

The TLB miss handler can use the context register immediately, and the handler looks much like the following (those who know the MIPS instruction set will notice that a few NOPs have been omitted for clarity):

```
mfc0 k0,tlbcxt     #move the contents of TLB
                   #context register into k0
mfc0 k1,epc        #move PC of faulting load
                   #instruction into k1
lw k0,0(k0)        #load thru address that was
                   #inTLB context register
mtc0 k0,entry_lo   #move the loaded value
                   #into the EntryLo register
tlbwr              #write entry into the TLB
                   #at a random slot number
j k1               #jump to PC of faulting
                   #load instruction to retry
rfe                #RESTORE FROM
                   #EXCEPTION
```

The handler first moves the address out of the TLB context register into general-purpose register k0, through which it can use the address as a load target. The program counter of the instruction that caused the TLB miss is found in exception register epc and is moved into general-purpose register k1. The handler loads the PTE into k0 then moves it directly into EntryLo. EntryHi is already filled in by the hardware; it contains the faulting VPN and the ASID of the currently executing process (the one that caused the TLB miss). The TLBWR instruction writes the PTE into the TLB at a randomly selected location.

At this point, the mapping for the faulting

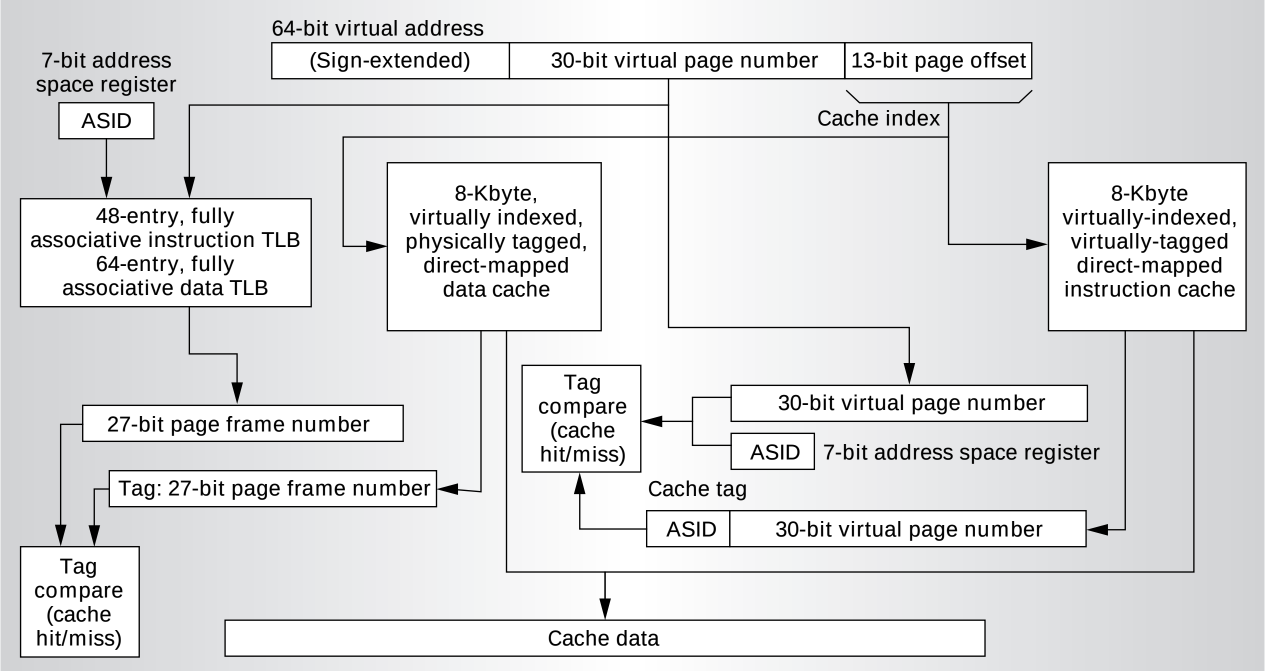
64-bit virtual address
(Sign-extended)
30-bit virtual page number
13-bit page offset
7-bit address space register
ASID
Cache index
48-entry, fully associative instruction TLB 64-entry, fully associative data TLB
8-Kbyte, virtually indexed, physically tagged, direct-mapped data cache
8-Kbyte virtually-indexed, virtually-tagged direct-mapped instruction cache
27-bit page frame number
Tag: 27-bit page frame number
Tag compare (cache hit/miss)
30-bit virtual page number
ASID
7-bit address space register
Cache tag
ASID
30-bit virtual page number
Tag compare (cache hit/miss)
Cache data

Figure 4. Alpha 21164 address translation mechanism. The virtual bits of the page offset effectively physically index each of the 8-Kbyte split caches. The data cache is physically tagged and the instruction cache is virtually tagged. The instruction TLB is accessed every reference to provide page protection.

address is in the TLB, and the hardware can retry the faulting instruction by jumping to its address (previously moved from epc to k1). MIPS uses delayed branches, so the instruction immediately after any branch instruction executes with the branch. Thus, the Restore From Exception instruction executes before the target instruction of the jump. The Restore returns the processor (which runs in kernel mode during exceptions) to user mode.

Alpha

Alpha (Figure 4) is a 64-bit architecture with a memory management design similar to MIPS. Applications generate 64-bit pointers, of which anywhere from 43 to 64 bits are translated, depending on the processor implementation. Alpha implementations that support less than 64 bits of addressing must ensure that the high-order bits of a virtual address are sign-extended from the most significant translated bit.

The Alpha architecture has split TLBs and split caches, and, as in the MIPS architecture, software manages the TLBs. However, the OS does not have direct access to the TLB but indirect access through the PALcode—the privileged access library. For example, PALcode, not OS code, loads PTEs from a hardware register (like EntryHi/EntryLo in MIPS) into the TLB. Though software manages the TLB, the TLB replacement policy in the 21164 is fixed at not-most-recently-used—the OS does not define it. The 21164 supports a fixed 8-Kbyte page size, a 43-bit virtual address, and a 40-bit physical address.

Virtual and physical address spaces

OSF/1, one of Alpha's main OSs, divides the virtual space into three regions: seg0 is the bottom half, seg1 is the top quarter, and kseg is the middle quarter. The regions are divided by the two most significant bits of the translated address, as opposed to the MIPS division, which is based on the high-order bits of the 64-bit address, regardless of how many bits the hardware translates.

Seg0 and seg1 comprise the user space and are mapped through the TLBs. Kseg maps directly onto the physical address space by zeroing the top two bits. Note that the kernel may use seg0 and seg1 for its own virtual data structures (the page tables). While the MIPS architecture distinguishes between user-level virtual

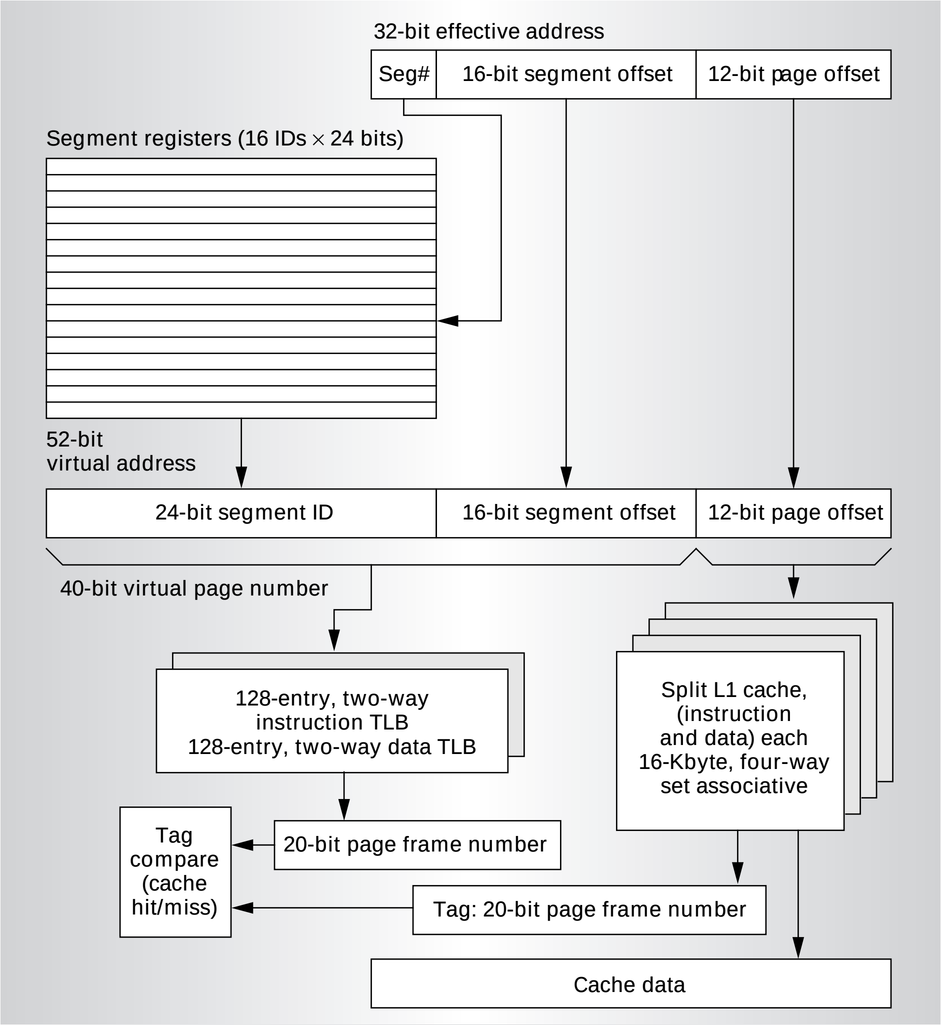

Figure 5. PowerPC 604 address translation mechanism. Each 16-Kbyte, four-way set-associative cache uses the page offset as a cache index, effectively making the cache physically indexed. A block address translation mechanism occurs in parallel with the TLB access, but is

access and kernel-level virtual access by the top bit of the virtual address, the Alpha makes no such distinction. The OS manages the division of the virtual space through the page tables.

Unlike MIPS, which uses the top bits of the virtual address to demarcate regions of differing memory behavior, Alpha's physical memory is divided into regions of different behavior to support features like cacheable and noncacheable memory. Physical space is divided by the top two bits of the physical address into four regions, each with potentially different behavior. The behavior is left to the implementation to define; possibilities include "normal" memory, I/O space, and noncached uncacheable memory, among others.

Superpages and shared memory

The Alpha TLB entry includes such fields as address-space-match, similar to the global bit in the MIPS TLB; if the bit is set, the TLB entry matches positively for all ASIDs. The flags also include granularity-hint (GH), a two-bit field that supports superpages. GH indicates that the entry maps a set of 8^{GH} contiguous pages, that is, a block of 1, 8, 64, or 512 pages.

The 21164 uses 7-bit address space identifiers; this scheme has the same need for flushing as the MIPS architecture due to a small number of contexts. The 21164 split-cache sizes are 8 Kbytes each, which is the processor's page size, making the caches effectively indexed by the physical address. Thus the caches have the access time of a virtually indexed cache but the cache-consistency behavior of a physically indexed cache.

PowerPC

The PowerPC (Figure 5) features a hardware-defined inverted page table structure, a hardware-managed TLB miss mechanism, block address translation for superpages, and a 52-bit segmented virtual address space for 32-bit implementations of the architecture. The 64-bit PowerPC implementations have an 80-bit segmented virtual address space. The block address translation mechanism is not shown in the figure—it is performed in parallel with the pictured translation mechanism. It is similar to TLB translation, except that the effective address, not the extended virtual address, references the entries.

Segmented address translation

PowerPC's memory management uses paged segmentation; the TLBs and page table map an extended virtual address space, not

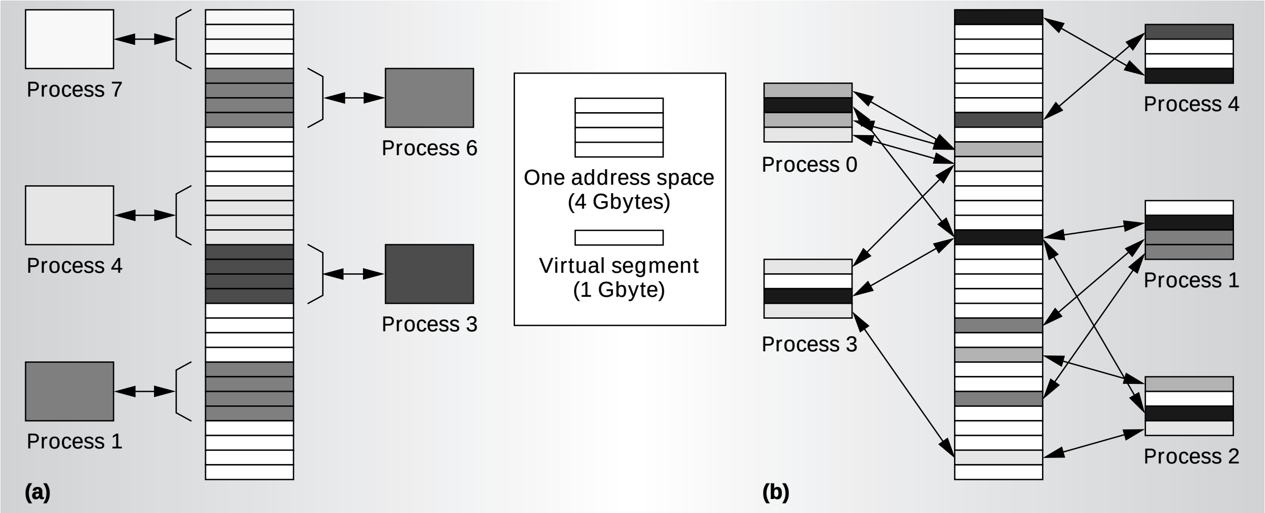

Figure 6. Difference between ASIDs and segmentation in two virtual addressing schemes. A 3-bit ASID extends the 32-bit address space (a). A 35-bit segmented address space is divided into fixed-length, 1-Gbyte segments (b). In both schemes, processes generate 32-bit virtual addresses, and the hardware translates a 35-bit virtual address.

the application's effective address space. PowerPC segments are 256-Mbyte continuous regions of virtual space; 16 segments comprise an application's address space; and the global space contains 2^{24} segments.

Sixteen segment registers map the user address space onto the segmented global space. The top four bits of an effective address index the segment registers to obtain a 24-bit segment identifier. This segment ID replaces the top four bits of the 32-bit effective address to form a 52-bit virtual address. The 64-bit PowerPC implementations use the top 36 bits of the effective address to index the segment registers. This corresponds to an enormous table that would never fit in hardware. As a result, the segment mapping in 64-bit implementations uses a segment look-aside buffer (a fully associative cache much like a TLB) rather than a hardware table.

Figure 6 compares ASIDs with PowerPC-like segmentation using fixed-length segments. In Figure 6a, a 32-Gbyte space is organized into eight 4-Gbyte regions, each of which corresponds to exactly one process determined by the process's ASID. In Figure 6b, the global 32-Gbyte segmented space is an array of 1-Gbyte virtual segments. Each segment can map into any number of process address spaces at any location (or at multiple locations) within each process address space. The size of the ASID (in this case, eight processes) limits the number of processes in the Figure 6a scenario, while the number of processes in Figure 6b has a combinatorial limit.

The PowerPC does not provide explicit ASIDs; address spaces are protected through the segment registers, which only the OS can modify. The OS enforces protection by controlling the degree to which segment identifiers may be overlapped and can even share a segment safely between a small set of processes. This is in sharp contrast to most other sharing schemes in which processes and pages have but one access ID each, and in which the only way to support hardware access to shared pages with different access IDs is to mark the pages visible to all processes.

The 24-bit-wide segment identifiers support over a million unique processes on a system. Therefore, required cache or TLB flushing occurs very infrequently, assuming shared memory is implemented through segmentation.

Block address translation

To support superpages, the PowerPC defines a block address translation (BAT) mechanism that operates in parallel with the TLB. The BAT takes precedence over the TLB whenever the BAT mechanism signals a hit on any given translation. Blocks can range in size from 128 Kbytes to 256 Mbytes. The mechanism uses a fully associative lookup on four BAT registers, each of which is similar to a TLB entry.

BAT registers contain information such as

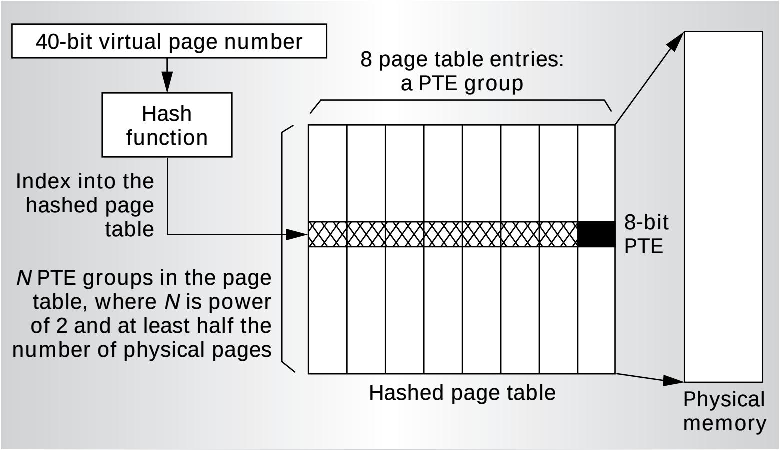
40-bit virtual page number
Hash function
Index into the hashed page table
N PTE groups in the page table, where N is power of 2 and at least half the number of physical pages
8 page table entries: a PTE group
8-bit PTE
Hashed page table
Physical memory

Figure 7. PowerPC hashed page table structure.

fine-grained protection (whether the block is readable and/or writable) and storage access control (whether the block is write-through and/or caching-inhibited, and/or needs to be maintained as cache-coherent). A BAT register provides no address-space protection besides making a block available only to privileged (kernel) access. If a block is available to a single user-level process, all user processes share it. In this case, if sharing is not desired, the BAT register contents need to be flushed on a context switch. The PowerPC 604 has two sets of BAT registers: one for the instruction stream and one for the data stream.

Hashed page table

The PowerPC architecture defines a hashed page table, pictured in Figure 7. A variation on the inverted page table, it is an eight-way set-associative software cache for PTEs,[11] walked by hardware. The organization scales well to a 64-bit implementation—inverted tables scale with the size of the physical address space, not the virtual address space. However, since the table is not guaranteed to hold all active mappings, the OS must manage a backup page table as well.

The design is similar to the canonical inverted page table, except that it eliminates the hash anchor table (which reduces the number of memory references by one), and the fixed-length PTE group replaces the collision chain. The table is eight PTEs wide. If more than eight VPNs hash to the same PTE group, any extra PTEs are simply left out or placed into a secondary PTE group. On a TLB miss, the hardware loads an entire PTE group and performs a linear search for a matching virtual address. If the PTE is not found in the first PTE group, the hardware performs a second lookup to a different PTE group, based on a secondary hash value (the one's complement of the primary hash value).

Since the location of a PTE in the table bears no relation to either the VPN or the PFN in the mapping, both VPN and PFN must be stored in the PTE. Each PTE is therefore 8 bytes wide. In 64-bit implementations, each PTE is 16 bytes wide.

PA-RISC 2.0

The PA-RISC 2.0 (Figure 8) is a 64-bit design with a segmentation mechanism similar to a PowerPC's. Processes generate 64-bit addresses that are mapped through space registers to form 96-bit virtual addresses translated by the TLB and page table. The architecture differs from the PowerPC by its addition of two types of page-access keys: the protection ID, which identifies a running process, and the access ID, which identifies a virtual page. The architecture allows user-level processes to change the contents of the space registers, enabling an application to extend its address space at will. This user-level access to the space registers creates the need for the additional protection mechanism.

Spaces and space registers

Every 64-bit user address combines with a 64-bit space ID to create a 96-bit virtual address that references the TLB and virtual caches. The space IDs are held in eight space registers that are selected by fields within the instruction or by the top bits of the effective address. As in the PowerPC, the space IDs are concatenated with the user's effective address, but the mechanism is designed to allow a variable-size segment.

Unlike the PowerPC segment identifier, which replaces the top four bits, the space ID in the PA-RISC replaces anything from the top 2 bits to the top 32 bits. The top 34 bits of the 96-bit virtual address come from the space ID; the bottom 32 bits come from the user's effective address; and the middle 30 bits are the logical OR of the 30-bit overlap of the two. This allows the OS to define a segmentation granularity. A logical OR suggests a

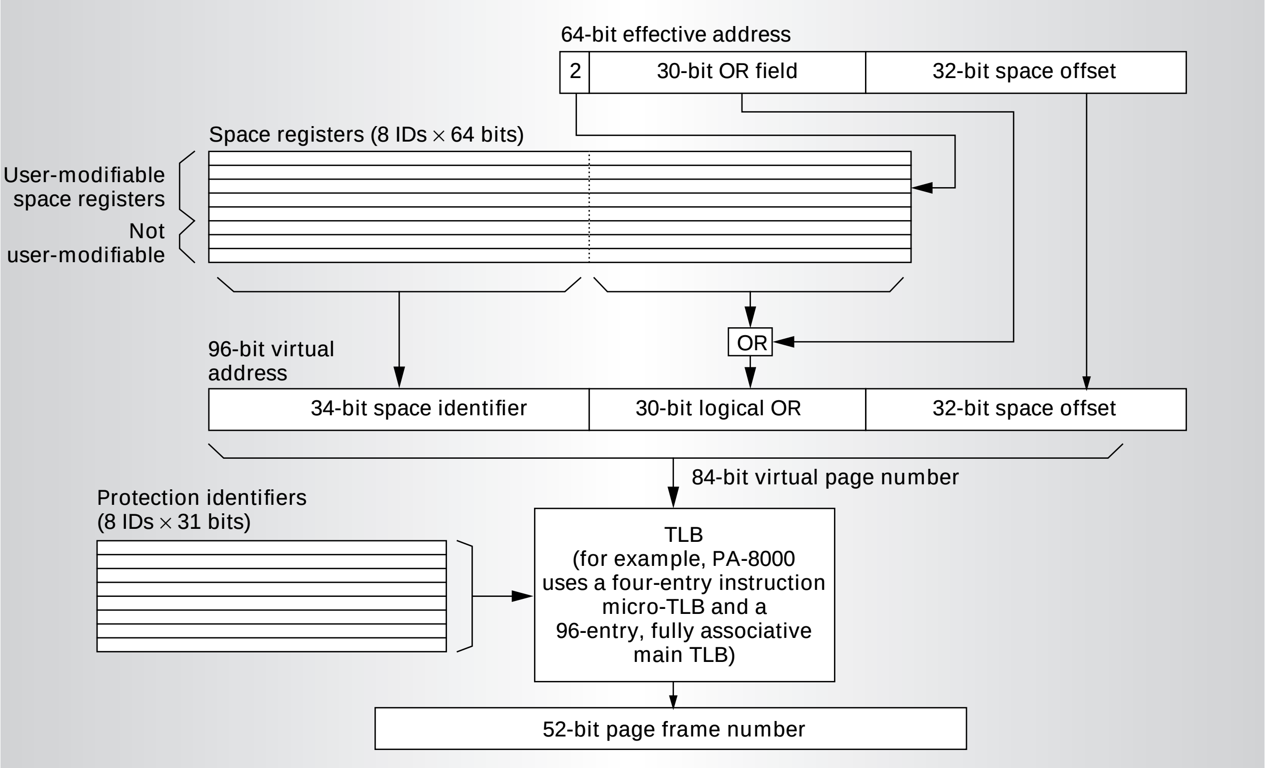
64-bit effective address
2
30-bit OR field
32-bit space offset
Space registers (8 IDs × 64 bits)
User-modifiable space registers
Not user-modifiable
96-bit virtual address
OR
34-bit space identifier
30-bit logical OR
32-bit space offset
84-bit virtual page number
Protection identifiers (8 IDs × 31 bits)
TLB (for example, PA-8000 uses a four-entry instruction micro-TLB and a 96-entry, fully associative main TLB)
52-bit page frame number

Figure 8. PA-RISC 2.0 address translation mechanism. The architecture uses large virtual caches located off chip.

simple and effective organization; one can choose a partition (for example, half way, at the 15-bit mark), and allow processes to generate only virtual addresses with the top 15 bits of this 30-bit field set to 0. Similarly, all space IDs would have the bottom 15 bits set to 0, so that the logical OR would effectively yield a concatenation. This example gives us a PowerPC segment-like concatenation of a (34 + 15 = 49-bit) space ID with the (32 + 15 = 47-bit) user virtual address.

The top two bits of the effective address select among four of the eight space registers. The rest of the registers are accessed rarely and only by the OS.

Naming, protection, and access keys

PA-RISC is unique in its address-space organization and protection. In most other architectures, address-space protection is guaranteed through naming. That is, addresses are protected from a process by preventing that process from generating the appropriate virtual address. For instance, in Figure 6a, processes in ASID-based systems are prevented from generating addresses outside of their portion of the global space. The naming mechanism—appending an ASID to every virtual address—prevents processes from generating addresses corresponding to locations in other address spaces. Even systems based on the segmentation scheme in Figure 6b can use naming for protection. For example, the PowerPC segmentation mechanism allows any process to generate any address within the global space, but only the OS can put specific values into the segment registers. This effectively restricts any given process from generating addresses outside of its 32-bit space. The IA-32 segmentation mechanism is very similar.

The PA-RISC mechanism differs from other segmented architectures in a simple but important way: it does not prevent a user-level application from modifying the contents of the space registers. Thus, a process can produce any virtual address in the 96-bit global space without requiring permission from the OS. The PA-RISC, unlike the other schemes, does not use naming to provide protection, and so it must provide protection through other means. The PA-RISC solution is to tag every page with a 31-bit access identifier, roughly corresponding to

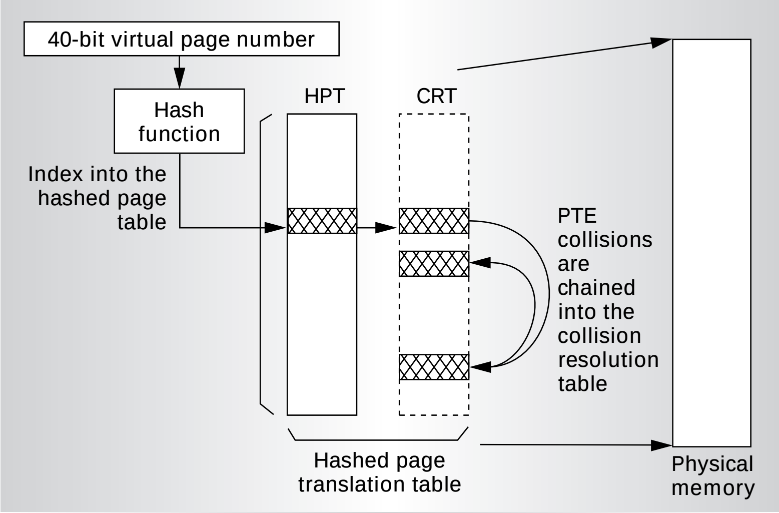
40-bit virtual page number
Hash function
Index into the hashed page table
HPT
CRT
PTE collisions are chained into the collision resolution table
Hashed page translation table
Physical memory

Figure 9. PA-RISC hashed page translation table (HPT). When different VPNs produce the same hash value, the resulting PTE collisions are chained into the collision resolution table (CRT), which can be a separate data structure.

an ASID in other architectures. Instead of assigning a single ASID to every executing process, the PA-RISC assigns eight IDs; each running process has eight 31-bit protection identifiers associated with it. A process may access a page if any of its protection IDs match the page's access ID. As in the PowerPC, this allows a small set of user-level processes to access a given page, which differs markedly from the all-or-nothing sharing approach found in most processors.

Hashed page translation table

While the architecture does not define a particular page table organization, PA-RISC engineers have published a variant of the inverted page table.[12] The hashed page translation table is pictured in Figure 9. As in the PowerPC, it dispenses with the hash anchor table, eliminating one memory reference on every PTE lookup; the table can hold more entries than physical pages in the system. The collision chain is held in the page table or in a separate structure called the collision resolution table. The PFN cannot be deduced from an entry's location in the page table, so each PTE contains both the VPN and the PFN for the mapped page.

This mechanism is implemented in hardware in the PA-7200, which performs a single probe of the table in hardware and defers to software if the initial probe fails. Thus, the hashed table can act as a software TLB[11] fronting a main page table of any organization, even a hierarchical table.[12]

UltraSPARC

The UltraSPARC (Figure 10) is another 64-bit design. Like the MIPS and Alpha designs, each implementation does not recognize all of the 64 bits; in the UltraSPARC-I the top 20 bits must be sign-extended from the 44th bit. The implementation sets the size of the physical address. The UltraSPARC memory management organization is notable for the way it uses ASIDs, called ASIs in Sun terminology.

ASIs

These are essentially opcodes to the MMU. The 8-bit ASI is not used to identify different contexts directly, but to identify data formats and privileges and to reference indirectly one of a set of context identifiers. The following are a few of the basic ASIs reserved and defined by the UltraSPARC architecture; individual implementations may add more.

Non-restricted:	
ASI_PRIMARY	{_LITTLE}
ASI_PRIMARY_NOFAULT	{_LITTLE}
ASI_SECONDARY	{_LITTLE}
ASI_SECONDARY_NOFAULT	{_LITTLE}
Restricted:	
ASI_NUCLEUS	{_LITTLE}
ASI_AS_IF_USER_PRIMARY	{_LITTLE}
ASI_AS_IF_USER_SECONDARY	{_LITTLE}

Only processes operating in privileged mode may generate the restricted ASIs; user-level processes can generate nonrestricted ASIs. Any ASI labeled PRIMARY refers to the context ID held in the primary context register; any ASI labeled SECONDARY refers to the context ID held in the secondary context register; and any ASI labeled NUCLEUS refers to the context ID held in the nucleus context register. The {_LITTLE} suffixes indicate that the target of the load or store should be interpreted as a little-endian object; otherwise the MMU treats the data as big-endian. The NOFAULT directives tell the MMU to load the data into the cache and to fail silently if the data is not actually in memory. This function can be used for speculative prefetching.

The default ASI is ASI_PRIMARY; it indi-

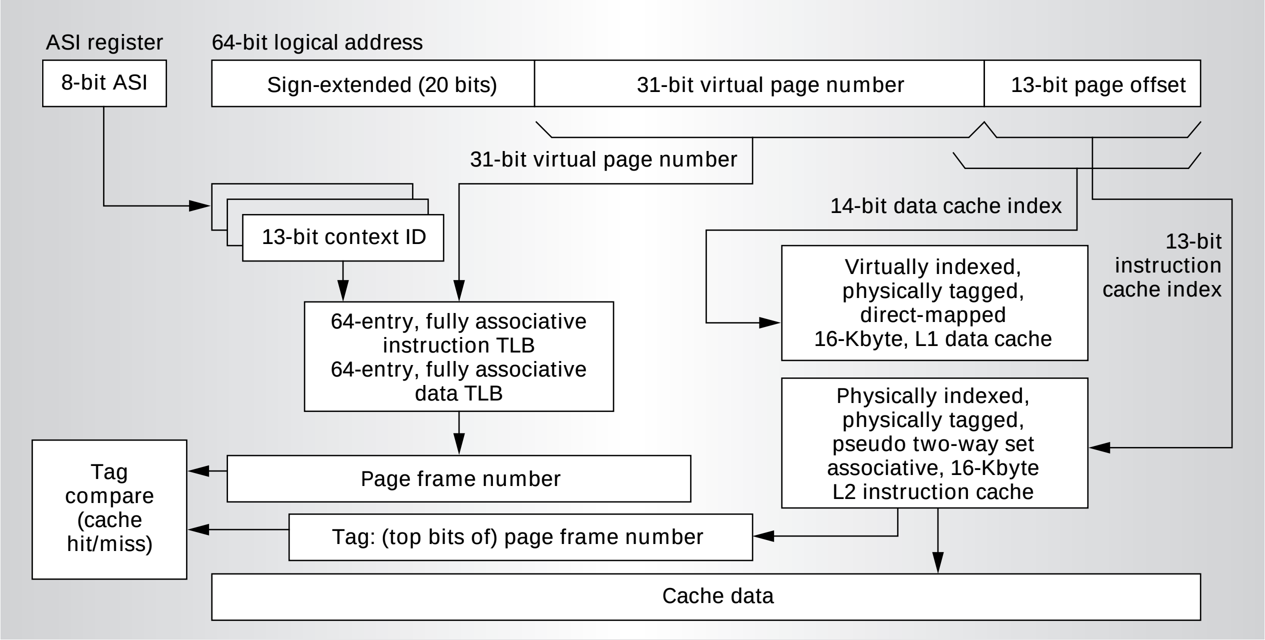
ASI register
8-bit ASI
64-bit logical address
Sign-extended (20 bits)
31-bit virtual page number
13-bit page offset
31-bit virtual page number
14-bit data cache index
13-bit context ID
13-bit instruction cache index
Virtually indexed, physically tagged, direct-mapped 16-Kbyte, L1 data cache
64-entry, fully associative instruction TLB 64-entry, fully associative data TLB
Physically indexed, physically tagged, pseudo two-way set associative, 16-Kbyte L2 instruction cache
Tag compare (cache hit/miss)
Page frame number
Tag: (top bits of) page frame number
Cache data

Figure 10. UltraSPARC-I address translation mechanism. The instruction cache is pseudo two-way set-associative (the most significant bit of the cache index is predicted), so only the page offset bits are needed to index the cache. The instruction cache is effectively indexed by the physical address, and there are no cache synonym problems. The direct-mapped data cache is twice the page size; it is indexed by the virtual address, and synonym problems can occur. As a solution, the OS aligns all aliases on 16-Mbyte boundaries, which is the largest L2 cache the architecture supports.

cates to the MMU that the current user-level process is executing. When the OS executes, it runs as ASI_NUCLEUS and can peek into the user's address space by generating PRIMARY or SECONDARY ASIs (the AS_IF_USER ASIs).

System software can move values in and out of the MMU's context registers that hold the PRIMARY and SECONDARY context IDs. This supports shared memory and resembles the PA-RISC space IDs and protection IDs in its ability to give applications simultaneous access to multiple protection domains.

For instance, user-level OS servers that peek into the address spaces of normal processes can use the ASI_SECONDARY identifier. A user-level dynamic linker or server implementing a particular OS API, as in Mach or Windows NT, must set up and manage the address spaces of processes running under its environment. It could run as PRIMARY and move the context ID of the child process into the secondary context register. It could then explicitly use SECONDARY ASIs to load and store data values to the child's address space, and use PRIMARY ASIs to execute its own instructions and reference data in its own address space.

Processes can generate ASIs either directly or indirectly. Certain instructions, such as the Load/Store From Alternate Space as well as atomic memory-access instructions like Swap Register Memory and Compare and Swap Word, each contain an ASI in an immediate field, or reference the ASI register. These instructions directly specify an ASI whenever an immediate bit is zero, indicating that the ASI is to be taken from another immediate field within the instruction. When the bit is 1, the ASI is taken from the ASI register. Other instructions can only generate ASI_PRIMARY, the default ASI.

IA-32

The Intel architecture (Figure 11, next page) is one of the older memory management architectures manufactured today. It is also one of the most widely used and one of the most complex. Its design is an amalgam of several techniques, making it difficult to cover the entire organization briefly. We therefore complement the description of Intel's system call and protection mechanisms in Hennessy and Patterson[13] by describing the

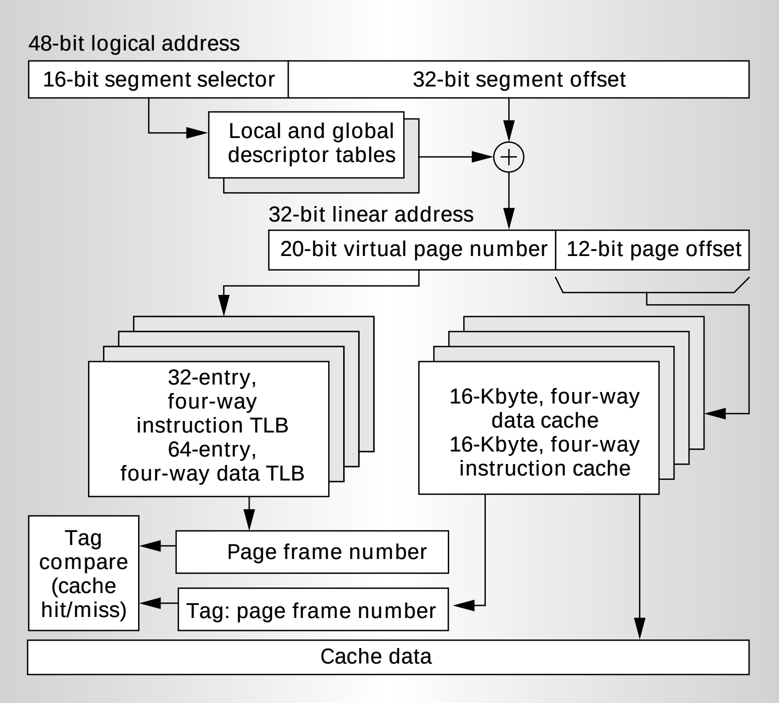
48-bit logical address
16-bit segment selector
32-bit segment offset
Local and global descriptor tables
+
32-bit linear address
20-bit virtual page number
12-bit page offset
32-entry, four-way instruction TLB 64-entry, four-way data TLB
16-Kbyte, four-way data cache 16-Kbyte, four-way instruction cache
Tag compare (cache hit/miss)
Page frame number
Tag: page frame number
Cache data

Figure 11. The Pentium II address translation mechanism. Each 16-Kbyte, four-way set-associative cache uses the page offset as a cache index, effectively making the cache physically indexed. The architecture defines separate block TLBs to hold superpage mappings, but they are not shown.

Pentium II's address translation mechanism.

The Pentium II memory management design features a segmented 32-bit address space, split TLBs, and a hardware-managed TLB miss mechanism. Processes generate 32-bit near pointers and 48-bit far pointers that are mapped by the segmentation mechanism onto a 32-bit linear address space. The linear address space is mapped onto the physical space through the TLBs and hardware-defined page table. The canonical two-tiered hierarchical page table uses a 4-Kbyte root-level table, 4-Kbyte PTE pages, and 4-byte PTEs.

The processor supports 4-Kbyte and 4-Mbyte page sizes, as well as 2-Mbyte superpages in some modes. Physical addresses in the IA-32 are 32 bits wide, though the Pentium II supports an even larger physical address through its physical address extension or 36-bit page-size extension modes. In either of these modes, physical addresses produced by the TLB are 36 bits wide.

Protection and shared memory

The Pentium II is another segmented architecture with no explicit ASIDs. For performance reasons, its segmentation mechanism is often unused by today's operating systems, which typically flush the TLBs on context switch to provide protection. The caches are physically indexed and tagged and therefore need no flushing on context switch, provided the TLBs are flushed.

The location of the root page table is loaded into one of a set of control registers, CR3, and on a TLB miss the hardware walks the table to refill the TLB. If every process has its own page table, the TLBs are guaranteed to contain only entries belonging to the current process—those from the current page table—if the TLBs are flushed and the value in CR3 changes on context switch. Shared memory is often implemented by aliasing—duplicating mapping information across page tables.[1]

Writing to the CR3 control register flushes the entire TLB; during a context switch, the hardware writes to CR3, so flushing the TLB is part of the hardware-defined context-switch protocol. Globally shared pages (protected kernel pages or library pages) can have the global bit of their PTE set. This bit keeps the entries from being flushed from the TLB; an entry so marked remains indefinitely in the TLB until it is intentionally removed.

Segmented addressing

The IA-32 segmentation mechanism supports variable-size (1-byte to 4-Gbyte) segments; the size is set by software and can differ for every segment. Unlike other segmented schemes, in which the global space is much larger than each process's virtual space, the IA-32 global virtual space (the linear address space) is the same size or, from one viewpoint, smaller than an individual user-level address space.

Processes generate 32-bit addresses that are extended by 16-bit segment selectors. Two bits of the 16-bit selector contain protection information, 13 bits select an entry within a software descriptor table (similar to the PowerPC segment registers or the PA-RISC space registers), and the last bit chooses between two different descriptor tables.

Conceptually, an application can access several thousand segments, each of which can range from 1 byte to 4 Gbytes. This may seem to imply an enormous virtual space, but a process's address space is actually 4 Gbytes. During address generation, the segment's base address is added to the 32-bit address the process generates. A process actually has access to several thousand segments, each of which ultimately lies completely within the 4-Gbyte linear address space. The processor cannot distinguish more than four unique Gbytes of data at a time; it is limited by the linear address space.

The segment selector indexes the global and local descriptor tables. The entries in these tables are called segment descriptors and contain information including the segment's length, protection parameters, and location in the linear address space. A 20-bit limit field specifies the segment's maximum legal length from 1 byte to 4 Gbytes. A granularity bit determines how the 20-bit limit field is to be interpreted. If the granularity bit is clear, the limit field specifies a maximum length from 1 byte to 1 Mbyte in increments of 1 byte. If the granularity bit is set, the limit field specifies a maximum length from 4 Kbytes to 4 Gbytes in increments of 4 Kbytes.

The segment descriptor also contains a 2-bit field specifying one of four privilege levels (highest is usually reserved for the OS kernel, lowest for user-level processes, and intermediate levels are for OS services). Other bits indicate fine-grained protection, whether the segment is allowed to grow (for example, a stack segment), and whether the descriptor allows interprivilege-level transfers. These transfers support special functions such as task switching, calling exception handlers, calling interrupt handlers, and accessing shared libraries or code within the OS from the user level.

Segment registers and pointers

For improved performance, the hardware caches six of a process's segment selectors and descriptors in six segment registers. The IA-32 divides each segment register into "visible" and "hidden" parts. Software can modify the visible part, the segment selector. The software cannot modify the hidden part, the corresponding segment descriptor. Hardware loads the corresponding segment descriptor from the local or global descriptor table into the hidden part of the segment register whenever a new selector is placed into the visible part of the segment register.

The segment registers are similar to the segment registers of the PowerPC architecture in that they hold the IDs of segments that comprise a process's address space. They differ in that they are referenced by context rather than by a field in the virtual address. Instruction fetches implicitly reference CS, the register holding the code segment selector. Any stack references (Push or Pop instructions) use SS, the register holding the stack segment selector. Destinations of string instructions like MOVS, CMPS, LODS, or STOS, implicitly use ES, one of the four registers holding data segment selectors. Other data references use DS by default, but applications can override this explicitly if desired, making available the remaining data-segment registers FS and GS.

A near pointer references a location within one of the currently active segments, that is, the segments whose selectors are in the six segment registers. On the other hand, the application may reference a location outside the currently active segments by using a 48-bit far pointer. This loads a selector and corresponding descriptor into a segment register; then, the segment is referenced. Near pointers are less flexible but incur less overhead than far pointers (one fewer memory reference), and so they tend to be more popular.

If used properly, the IA-32 segmentation mechanism could provide address-space protection, obviating the need to flush the TLBs on context switch. Protection is one of the original intents of segmentation.[3] The segments guarantee protection if the 4-Gbyte linear address space is divided among all processes, in the way that the PowerPC divides its 52-bit virtual space among all processes. However, 4 Gbytes is an admittedly small amount of space in which to work.

Superpages and large physical addresses

Note that an entry in the IA-32 root-level table maps a 4-Kbyte PTE page, which in turn maps 1,024 4-Kbyte user pages. Thus, a root-level PTE effectively maps a 4-Mbyte region. IA-32 supports a simple superpage mechanism: a bit in each root-level PTE determines whether the entry maps a 4-Kbyte PTE page or a 4-Mbyte user-level superpage directly. This reduces contention for TLB entries, and when servicing a TLB miss for a superpage, the hardware makes only one memory reference to obtain the mapping PTE, not two.

Normally, physical addresses in IA-32 are

32 bits. P6 processors offer two modes in which the physical address space is larger than 4 Gbytes: physical address extension and 36-bit page size extension. In either of these modes, physical addresses can be 36 bits wide. The physical address extension mode requires changes to the page table format. The PTEs expand to 8 bytes when this mode is enabled (double their normal size), and so the mode requires the addition of another level in the page table. On every TLB miss, the hardware must make three memory references to find the mapping PTE, not two. The mode also changes the semantics of the IA-32 superpage mechanism. In the reorganized IA-32 page table, a root-level PTE only maps a 2-Mbyte region; therefore superpages are 2 Mbytes when the physical address extension mode is enabled, not 4 Mbytes.

To limit confusion, designers added another mode that requires no changes to the page table or PTE size. The 36-bit page-size extension mode uses previously reserved bits in the PTEs to extend physical addresses. Therefore, with this mode the OS can support 4-Mbyte superpages and 36-bit physical addressing at the same time.

The use of one memory management organization over another has not catapulted any architecture to the top of the performance ladder, nor has the lack of any memory management function been the leading cause of an architecture's downfall. So, while it may seem refreshing to have so many choices of VM interface, the diversity serves little purpose other than to impede the porting of system software.

Designers have recognized this and have defined hardware abstraction layers[8,9] to hide hardware details from the OS; this simplifies both the transition to new versions of hardware platforms and the porting to entirely new architectures. Primitives in a memory-management abstraction layer include create/destroy_map, protect_page/region, wire_down_page/region, and so on. The types of mechanisms hidden by this software layer include TLB and data cache management instructions. However, these mechanisms cause little problem when porting system software; there are other underlying models that are so different from one another that they cannot be masked with a software layer. It is the hardware details that are most difficult to encapsulate in a hardware abstraction layer that cause most of the problems, for example,

- the use of virtual vs. physical caches
- protection methods (ASIDs, segments, multiple IDs)
- address-space organization (segmented/flat, page table support, superpage support).

These hardware features force designers to make decisions about the high-level organization of the OS. Designers must make these decisions early on in the design process, and the decisions are difficult to undo when the OS has been implemented. For example, a virtual cache forces the OS to be aware of data-consistency issues. The hardware protection mechanism influences the OS's implementations of interprocess communication, memory-mapped files, and shared memory. The address-space organization mechanisms influence the OS's model of object allocation—how easy is it to allocate and destroy large/small objects, whether objects can have different characteristics (such as protection, cache coherency) based on the process accessing them, and how sparsely populated the address space can be. Clearly, it is very difficult to encapsulate these effects in a transparent software layer.

We thus have two choices: to live with diversity that serves no significant purpose or to standardize on support for memory management. Within standardization, there are further choices, including the elimination of most hardware support for memory management so that the software can define support,[14] or decision by fiat such as the adoption of the already de facto standard, the IA-32 (or a subset thereof). The advantage of adopting the IA-32 interface is the large number of hardware and software developers already familiar with the interface, as well as its relatively good performance. The disadvantage is that it might not scale well to 64-bit architectures. If one were starting with a clean slate, there is evidence to recommend a hardware-walked inverted page table,[16] which would resemble both IA-32 and PA-RISC, and would offer good performance and scale well to 64-bit address spaces and beyond. MICRO

Acknowledgments

We thank the many reviewers who helped shape this article, especially Joel Emer, Jerry Huck, Mike Upton, and Robert Yung for their comments and insights into the workings of the Alpha, PA-RISC, IA-32, and SPARC architectures. The Defense Advanced Research Projects Agency under DARPA/ARO contract DAAH04-94-G-0327 partially supported this work.

References

1. B.L. Jacob and T.N. Mudge, "Virtual Memory: Issues of Implementation," *Computer,* Vol. 31, No. 6, June 1998, pp. 33-43.
2. D.W. Clark and J.S. Emer, "Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement," *ACM Trans. Computer Systems,* ACM, New York, Vol. 3, No. 1, Feb. 1985, pp. 31-62.
3. E.I. Organick, *The Multics System: An Examination of Its Structure,* MIT Press, Cambridge, Mass., 1972.
4. *Pentium Processor User's Manual,* Intel Corporation, Mt. Prospect, Ill., 1993.
5. G. Kane and J. Heinrich, *MIPS RISC Architecture,* Prentice-Hall, Englewood Cliffs, N.J., 1992.
6. J.E. Smith, G.E. Dermer, and M.A. Goldsmith, Computer System Employing Virtual Memory, patent 4,774,659, US Patent Office, Wash., D.C., Sep. 1988.
7. D. Nagle et al., "Design Trade-Offs for Software-Managed TLBs," *ACM Trans. Computer Systems,* Vol. 12, No. 3, Aug. 1994, pp. 175-205.
8. R. Rashid et al., "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," *IEEE Trans. Computers,* Vol. 37, No. 8, Aug. 1988, pp. 896-908.
9. H. Custer, *Inside Windows NT,* Microsoft Press, Redmond, Wash., 1993.
10. J. Liedtke, "On Micro-Kernel Construction," *SOSP-15,* ACM, Dec. 1995, pp. 237-250.
11. K. Bala, M.F. Kaashoek, and W.E. Weihl, "Software Prefetching and Caching for Translation Lookaside Buffers," *Proc. OSDI-1,* Usenix Corporation, Berkeley, Calif., Nov. 1994, pp. 243-253.
12. J. Huck and J. Hays, "Architectural Support for Translation Table Management in Large Address Space Machines," *Proc. ISCA-20,* IEEE Computer Society, Los Alamitos, Calif., May 1993, pp. 39-50.
13. J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach,* 2nd ed., Morgan Kaufmann Publishers, Inc., San Francisco, Calif., 1996.
14. B.L. Jacob and T.N. Mudge, "Software-Managed Address Translation," *Proc. HPCA-3,* IEEE CS Press, Feb. 1997, pp. 156-167; http://computer.org/conferen/hpca97/77640156.pdf.
15. B.L. Jacob and T.N. Mudge, "A Look at Several Memory Management Units, TLB-Refill Mechanisms, and Page Table Organizations," *ASPLOS-8,* ACM, Oct. 1998, to appear.

Bruce Jacob is an assistant professor of electrical and computer engineering at the University of Maryland, College Park. His present research includes the design of memory management organizations and hardware architectures for real-time and embedded systems.

Jacob received the AB degree in mathematics from Harvard University, and the MS and PhD degrees in computer science and engineering from the University of Michigan, Ann Arbor. He is a member of the IEEE and the ACM, and has authored papers on computer architecture, analytical modeling, distributed systems, astrophysics, and algorithmic composition.

Trevor Mudge is a professor of electrical engineering and computer science at the University of Michigan, Ann Arbor, and director of the Advanced Computer Architecture Laboratory, a group of eight faculty and 70 graduate research assistants. He also consults for several computer companies. His research interests include computer architecture, computer-aided design, and compilers.

Mudge received the BSc degree in cybernetics from the University of Reading, England, and the MS and PhD degrees in computer science from the University of Illinois, Urbana. He is an associate editor for *IEEE Transaction on Computers* and *ACM Computing Surveys*, a Fellow of the IEEE, and a member of the ACM, the IEE, and the British Computer Society.

Direct comments about this article to Bruce Jacob, Dept. of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742; blj@eng.umd.edu.