

HP, INTEL COMPLETE IA-64 ROLLOUT

Virtual Memory, Interrupts More Conventional Than ISA

By Keith Diefendorff {4/10/00-01}

HP and Intel have finally finished the long-drawn-out task of revealing the IA-64 architecture, a process that began more than two years ago at the 1997 Microprocessor Forum. Speaking at Intel's spring developer forum (IDF), IA-64-architects Jerry Huck and Rumi Zahir laid out

the details of the IA-64 system architecture, including descriptions of its virtual-memory mechanisms, memory protection, multiprocessor coherence, and interrupt structure.

Unlike the unconventional approach HP and Intel took with IA-64's VLIW instruction-set architecture (ISA) (see [MPR 5/31/99-01](#), "IA-64: A Parallel Instruction Set"), the companies took a far less radical tack on system architecture. Borrowing liberally from existing RISC processors, especially from PA-RISC and PowerPC, IA-64 processors will require little redesign of operating-system internals.

Although the vendors released system-architecture details to selected IA-64 software developers under nondisclosure agreements some time ago, the recent IDF disclosure allows any company to create any level of software, right down to and including operating-system kernels and platform BIOSs. With this step, HP and Intel have now completed paving the way for the first Itanium processors to enter the market, an event we expect to occur during the second half of this year. Itanium (née Merced) silicon has been in the hands of anointed software developers since late last year, and at the International Solid-State Circuits Conference (ISSCC) Intel revealed that the silicon will be introduced at a speed of 800MHz (see [MPR 2/28/00-msb](#), "Itanium Meets 800MHz Goal"), as we had previously anticipated.

VM Supports MAS and SAS

Operating systems are generally built on one of two virtual-memory models: the multiple-address space (MAS) model—

implemented by Windows NT, Linux, BSD, VMS, and Mach-based versions of Unix—or the single-address-space (SAS) model used in proprietary versions of Unix, such as HP-UX and IBM's AIX. (The SAS model is also referred to as the global-address-space model.) Historically, processors have supported one model or the other, making them more or less adaptable to a given OS.

MAS operating systems typically use unique process identifiers (PIDs) and simple multilevel-indexed page-table

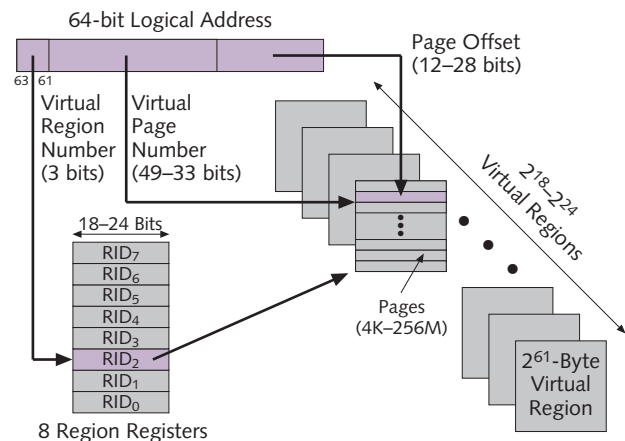


Figure 1. An IA-64 process is mapped onto a 2^{61} -byte virtual address space through eight region registers. This mapping supports either single-address-space or multiple-address-space views of virtual memory.

structures that isolate processes into separate address spaces. SAS-based OSs, in contrast, allocate all tasks into a single, very large virtual-address space, typically mapped by an inverted or hashed page table. The choice of model is largely one of OS-designer preference and of the target market for the OS. The MAS model tends to be simpler and somewhat more efficient for small systems, while the SAS model tends to be favored for large multiprocessor systems, because it makes sharing data among tasks more straightforward.

Because the processor hardware that is required to support each model is somewhat different, porting a MAS-based OS to a SAS-flavored processor, or vice versa, requires the OS to go through a number of contortions to force-fit its natural data structures to the dissimilar hardware. Reworking the OS to conform to the opposite flavor of processor is generally impractical, and the contortions can introduce a significant performance burden.

With Microsoft (and Windows NT) on one side and its partner HP (and HP-UX) on the other, Intel—in its inimitable fashion—took the obvious way out: it implemented both models. This schizophrenic approach, however, is probably justified in this case. Taking this approach makes IA-64 processors OS agnostic—which fits nicely into Intel’s plan of conquering the entire world with IA-64. And the hardware cost for supporting both address-space models is not large; while the MAS and SAS models are different in critical respects, the actual hardware difference is small. As a result, support for both models is unlikely to have any substantive adverse effects on cycle time or die area. Besides, the burden IA-64 bears of supporting the IA-32 (x86) memory-mapping model probably swamps the incremental cost associated with dual MAS/SAS hardware.

Processes See Flat 2⁶⁴-Byte Space

As on other 64-bit RISC systems, processes that run on IA-64 systems see a simple, flat 2⁶⁴-byte ($\approx 2 \times 10^{19}$ -byte)

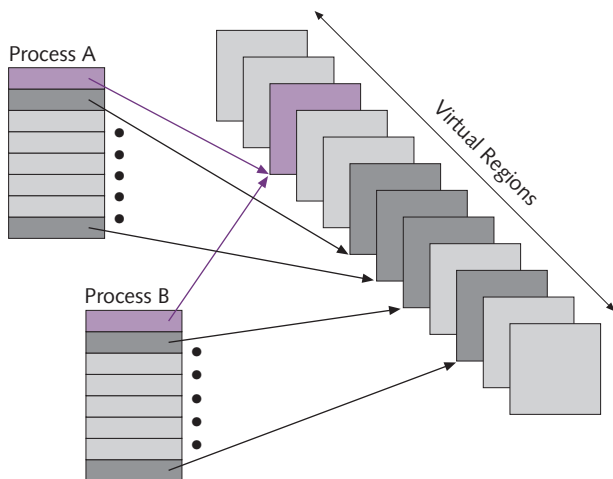


Figure 2. Two IA-64 processes can share data or instructions by having the OS allocate a common region (purple) to both processes.

logical-address space. Thus, IA-64 processes perform arithmetic on full 64-bit pointers and can reach any addresses within the enormous 2⁶⁴-byte space with a single load or store—without the overhead of manipulating segment registers. This logical-address space is over 4 billion times as large as that accessible by today’s IA-32-based (x86) processes.

Although such an enormous address space is massive overkill for today’s PC applications, addressability beyond the 2³²-byte ($\approx 4 \times 10^9$ -byte) limit of x86 processors is already important for large database applications. In fact, 64-bit addressing may soon become the ante to play in the large-server market that HP and Intel covet for initial IA-64 processors. It may eventually become important in other application areas as well; it does enable certain classes of algorithms that require a large address space but populate it only sparsely with data.

To support both the SAS and MAS models, the IA-64 system architecture defines virtual regions. Each processor implements a minimum of 2¹⁸ virtual regions and a maximum of 2²⁴, each 2⁶¹ bytes in size. Each process can have up to eight regions, which, as Figure 1 shows, are selected by the three most-significant bits (called the virtual-region number, or VRN) of a logical address.

The OS maps a process’s eight regions onto the system’s 2¹⁸ (or 2²⁴) possible regions through region identifiers (RIDs), which are loaded into eight hardware region registers (RRs) before the process executes. These registers are part of the process context and are saved and restored across process (context) switches. For SAS systems, RIDs can be interpreted as the most-significant address bits of a 2⁷⁹- (minimum) or 2⁸⁵- (maximum) byte global-virtual-address space. For MAS systems, RIDs are treated as address-space identifiers. Sharing between processes is facilitated by mapping shared regions into the RRs of multiple processes, as Figure 2 shows.

IA-64 supports 32-bit virtual addressing by three means: zero extension to 64 bits (used for all IA-32 accesses), sign extension to 64 bits (which requires software to ensure that the upper 32 bits of an address are always the same as bit 31), and pointer swizzling.

In the pointer-swizzling approach, the upper 2 bits of a 32-bit address select one of four virtual regions, and the entire 32-bit address (zero extended to 61 bits) is used as an offset into each of the four accessible regions. Swizzling the address bits in this manner supports either a flat 2³²-byte space (with a single RID) or a multiregion space.

The multiregion capability is important to ease the porting of old 32-bit programs onto new 64-bit IA-64 platforms. Using pointer swizzling, the compiler and OS can provide sharing and protection without requiring major surgery on the 32-bit source code to make it 64-bit safe. Sharing such things as dynamically linked libraries (DLLs) is a useful method of conserving precious resources, such as TLB entries.

Mapping to Physical Memory

At any point in time, a system can have many processes ready to run (with virtual memory allocated), although only a few are likely to be active (vying for CPU time), and only one can actually be running (occupying the CPU). To avoid having expensive physical memory (DRAMs) behind every byte of allocated virtual memory, IA-64 systems exploit temporal locality with demand paging, as do most modern systems. IA-64 pages are variable in size from 4K to 256M in 10 power-of-two increments (4K, 8K, 16K, 64K, 256K, 1M, 4M, 16M, 64M, and 256M). The architecture supports up to a 2^{63} -byte physical-address space, but current page-table definitions limit this space to 2^{50} bytes. (The 64th physical address bit is usurped as a cachable/uncachable memory attribute [described later] when virtual addressing is disabled.)

For mapping virtual pages onto physical pages, the IA-64 architecture defines a memory-based data structure called the virtual hashed page table, or VHPT. The VHPT has two possible organizations. The first, a simple per-region linear page table, is directly indexed by the virtual-page number (VPN); it uses a short-form (8-byte) entry. In this format, the per-region page table is part of the same region as the virtual address being translated. This organization will commonly be used by MAS systems. In MAS systems using the short format, the VHPT typically doubles as the OS page-table structure.

For SAS systems, IA-64 offers an alternative page-table organization. This scheme, which is similar to that used by PA-RISC and PowerPC, uses a hash function to index the VHPT (hence the name, virtual hashed page table). Conventional indexed page tables aren't suitable for the SAS model, because their size would be proportional to the size of virtual memory, which is enormous. Hashing, however, compacts the page table by taking advantage of the fact that the virtual-address space is sparsely occupied. This trick makes the page-table size proportional to the amount of implemented physical memory rather than to the much larger virtual-address space. In this form, IA-64 VHPT entries are 32 bytes in size.

VHPT entries are set up and maintained entirely by software. Although the VHPT could be searched by software for each translation, most IA-64 processors (including Itanium) will implement a hardware tablewalker to increase the search speed. This feature avoids draining the pipeline to run an out-of-line software tablewalk routine, and it allows asynchronous concurrent speculative translations.

While hardware tablewalkers are generally faster than software for servicing individual TLB misses, the IA-64 hardware tablewalker is limited to one of the two predefined page-table formats (per-region indexed or hashed). For some operating systems, however, neither of these organizations is optimal, and better overall performance can sometimes be obtained with a different structure, despite the lack of hardware tablewalk support. Therefore, motivated by a strong desire to achieve ubiquity with IA-64, HP and Intel

also provided mechanisms to support software tablewalking, allowing IA-64 to use any page-table format. To facilitate software tablewalking, the IA-64 architecture defines fast, vectored TLB-miss interruptions, as well as special instructions for manually loading and purging the TLBs.

TLB Includes Block-Address Translation

Even with a hardware tablewalker, however, performance would be unacceptable if every address translation required a search through a memory-based page table. So, like most modern processors, IA-64 processors implement translation-lookaside buffers to cache recent page translations on chip. The IA-64 architecture provides for separate instruction and data TLBs. Each TLB comprises two translation mechanisms: translation registers (TRs) and a translation cache (TC).

TC entries are similar to the traditional TLB entries found in most microprocessors; TRs are analogous to the block-address translation registers (BATs) of PowerPC. They are designed to efficiently map large, relatively static, areas of memory, thereby improving the utilization of precious TC entries, which usually map smaller, more-dynamic areas of memory. TRs, while they have the same format as TC entries, are managed strictly by software; that is, they are not automatically replaced and filled by the hardware tablewalker. TRs are filled by register number, using the insert-translation-register (*itr*) instruction, and they are purged according to virtual-address match using the purge-translation-register (*ptr*) instruction. Similar instructions (*itc* and *ptc*) are provided for software managing the TC.

Each IA-64 implementation is expected to have a minimum of eight instruction TRs and eight data TRs that are fully associative. The size and organization of the TCs are left to the discretion of the implementation, but every processor must have at least one data-TC entry and one instruction-TC entry. The architecture supports multilevel TCs, but only the first level is required to support all page sizes.

As Figure 3 indicates, memory accesses initiate an associative search for a matching virtual address across all TLB entries. The search attempts to locate an entry that matches both the virtual-page number (VPN) and the RID; including the region identifier eliminates the need to flush the TLB on a process switch.

On a TLB miss, the hardware tablewalker—if enabled—searches the memory-based VHPT and attempts to load an entry into the TC. In the event the tablewalker finds an empty VHPT entry (a miss) or detects a hash collision (VPN and RID do not match), it defers to software for a more exhaustive search of the operating system's primary page tables or of alternate VHPT collision chains (associativities). Also, if the tablewalker itself misses the TLB it will defer to the software VHPT-miss handler.

Address translations stall the pipeline for as long as it takes to resolve the physical address. In the case of a TLB hit, no stall is introduced; a TLB miss adds tens to hundreds of stall cycles, depending on which level of the memory

hierarchy contains the desired VHPT entry and whether a hardware or software tablewalk is performing the search. Software tablewalk, which is invoked by a TLB miss interruption, takes more cycles, because the instruction pipeline must be flushed, a software routing executed, and the pipeline restarted. On VHPT misses, which are always handled by software, many thousands of additional stall cycles can be added, depending on what action must be taken by the OS. In the worst case of a page fault, which can occur in any demand-paged virtual-memory system, many milliseconds can be required to bring a page into memory from the backing store on disk.

A Well-Protected System

The IA-64 system architecture defines elaborate facilities for protecting processes from inadvertent (or intentional) damage and for enforcing security policies among processes. Four protection mechanisms are provided: addressability, access rights, privilege level, and protection domains. Addressability is defined on a region basis, as previously described, and is enforced by manipulation of RIDs by trusted software (the OS). Appropriately configured, processes have no way to generate memory addresses that reference regions owned by other processes unless, of course, the OS intentionally grants shared access to a particular region.

Beyond addressability restrictions, each memory access is required to run a gamut of protection tests before it is permitted. The process must have an adequate privilege level, it must be attempting an allowed type of access (read, write, or execute), and the page being accessed must have a key that is capable of unlocking the corresponding protection domain. If any one of these protection mechanisms denies the access, a protection-violation fault is signaled to the operating system. All protection mechanisms are implemented with page granularity.

Four privilege levels are defined: 0, 1, 2, and 3. Level 0, generally reserved for use by the OS, is the most privileged, and it is the only level from which privileged instructions can be executed. A field in the processor status register (PSR.cpl) establishes the current privilege level. On each memory access, PSR.cpl is compared with the privilege level required by the page's TLB entry (TLB.pl). If PSR.cpl is less than or equal to TLB.pl, the access is permitted, as long as it is of a type (read, write, or execute) that is allowed by the access-rights field of the TLB.ar field in the TLB entry. Table 1 shows the allowed-access combinations as determined by the PSR.cpl, TLB.pl, and TLB.ar fields.

In addition to the access-rights and privilege-level protection, pages can also be tagged as belonging to different protection domains. Protection domains offer an efficient method for the OS to control access rights to groups of pages and, thereby, to improve the utilization of TLB entries. This feature is useful, for example, when implementing large object databases.

Domains are defined by protection-key registers (PKRs), of which there are at least 16 in every IA-64 processor. On each memory access, a key in the TLB entry associated with the referenced page is compared against all keys in the PKRs. The key size is implementation dependent, but it must be at least as large as the RID (18–24 bits). The matching entry, if one is found and marked valid, specifies the combination of access types (read, write, or execute) that is allowed to the domain. Since the OS controls the contents of the PKRs, it can quickly alter the access privileges of all the pages in a domain by changing the content of one protection-key register. Moreover, it can make this change without purging TLB entries (thereby increasing TLB utilization) and without having to modify the memory-based page-table entries of every affected page.

Protection domains are important features to SAS systems, due to the

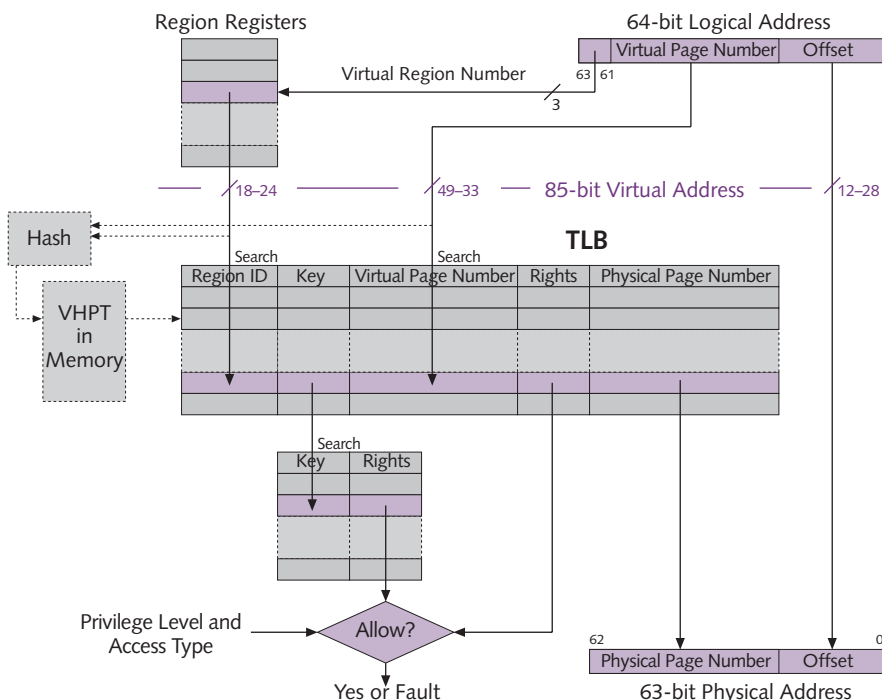


Figure 3. Like most modern processors, IA-64 processors translate virtual addresses to physical address through a page-based translation-lookaside buffer (TLB). Instruction TLBs and data TLBs both consist of at least eight block-address translation registers and an implementation-dependent number of translation-cache entries. The TRs and TC are associatively searched by virtual-page number and region ID. A TLB miss (dotted) causes an entry to be brought into the TLB from the memory-based VHPT. The TLB implements memory protection on the basis of privilege level, access rights, and protection domains (keys).

nature of the single global address space. MAS systems, however, rarely use protection domains and can disable protection-key checks using the PSR.pk bit in the processor-status register. This capability is in keeping with one of the design objectives set forth by IA-64 architects: to enable OS vendors to quickly port simple MAS systems to IA-64 processors and then gradually enable some of the more advanced SAS features, such as protection domains.

Attributes Restrict Flexibility When Necessary

The IA-64 architecture grants implementations a great deal of flexibility to cache, gather, reorder, or prefetch, as methods of optimizing memory performance. Sometimes, however, such activities create headaches for software, especially for OS software that is trying to achieve some particular objective. Therefore, the architecture provides a mechanism that software can use to restrict hardware's aggressive memory optimization activities.

To each virtual page, OS software can attach memory attributes that hardware will honor for all memory accesses

to that page. The attributes are set as a 3-bit field (TLB.ma) in the TLB entry for the page, and they control its cachability, write policy, sequentiality, speculation, and multi-processor coherency. Five different combinations of attributes are supported, as Table 2 shows.

Data and instructions in a page marked "cachable" can be freely copied into any level of the memory hierarchy. All cachable pages are coherent, meaning that the processor/memory system is responsible for ensuring that all processors within the coherence domain (tightly coupled multiprocessors) have a consistent view of memory. Coherence is enforced on the basis of physical addresses; virtual aliases (multiple virtual addresses targeting the same physical address) are supported, but performance can be degraded if aliased addresses are not at least 1M apart. Coherence between instruction and data caches is not required for IA-64 memory accesses, but it is required for IA-32 accesses. All cachable pages are free to use a writeback (copyback) write policy and to make speculative and out-of-order accesses.

Uncachable pages can be of two types: coalescing and noncoalescing. Coalescing pages support multiple stores to coalescing write buffers, which collect and merge store data so it can be written to memory in larger, more-bandwidth-efficient transactions. Similarly, loads may be merged together into larger read transactions. Coalescing pages are free to run loads and stores out of order; by definition, stores to coalescing buffers are performed out of order. Coalescing buffers are not required to be coherent, but a load to a coalesced page must see all prior stores from the same processor to the same page. The content of coalescing write buffers can be forced to memory with a flush-cache instruction (fc) that targets an address within 32 bytes of the data in the write buffer. Also, release operations (described later) have the side effect of forcing the contents of write buffers to memory. The architecture, however, provides no guarantee of when the writes must complete.

Uncachable noncoalescing pages are sequential and nonspeculative. Nonspeculative pages do not support speculative memory accesses, such as prefetches, advanced loads (ld.a), speculative loads (ld.s), or eager register-stack engine (RSE) spills and fills. However, to enable aggressive speculation, speculative loads to nonspeculative pages defer fault handling by suppressing the memory access and returning a deferred-exception indicator (NaT or NatVal) to the destination register. These indicators are simply propagated through subsequent operations, thereby deferring the actual interruption until the speculation is checked by a nonspeculative use or by a chk.s or a chk.a instruction.

The uncachable-exported attribute is similar to the uncachable attribute, but in some systems it provides an uncachable semaphore capability using the fetchadd instruction (described later). But this feature is optional, and Itanium processors do not support it.

Nonsequential pages allow load and store operations to be performed out of order, subject only to the architectural

TLB.ar	TLB.pl	Privilege Level (PSR.cpl)				Description
		3	2	1	0	
0	3	R	R	R	R	Read only
	2		R	R	R	
	1			R	R	
	0				R	
1	3	RX	RX	RX	RX	Read or execute
	2		RX	RX	RX	
	1			RX	RX	
	0				RX	
2	3	RW	RW	RW	RW	Read or write
	2		RW	RW	RW	
	1			RW	RW	
	0				RW	
3	3	RWX	RWX	RWX	RWX	Read or write or execute
	2		RWX	RWX	RWX	
	1			RWX	RWX	
	0				RWX	
4	3	R	RW	RW	RW	Read only and read or write
	2		R	RW	RW	
	1			R	RW	
	0				RW	
5	3	RX	RX	RX	RWX	Read or execute and read or write or execute
	2		RX	RX	RWX	
	1			RX	RWX	
	0				RWX	
6	3	RWX	RW	RW	RW	Read or write or execute and read or write
	2		RWX	RW	RW	
	1			RWX	RW	
	0				RW	
7	3	X	X	X	RX	Execute or promote and read or execute
	2	XP2	X	X	RX	
	1	XP1	XP1	X	RX	
	0	XP0	XP0	XP0	RX	

Table 1. IA-64 processes can run at one of four privilege levels, specified in the PSR. Each virtual page can be restricted to allow accesses from only certain privilege levels and by only certain types of memory requests. R = read, W = write, X = execute, Pn = promote to privilege level n.

requirement that read-after-write, write-after-write, and write-after-read dependencies to the same memory location be interlocked (executed in program order). Sequential pages, on the other hand, require that all loads and stores to the page be performed in the order specified by the program. The uncachable, nonspeculative, and sequential attribute would typically be specified for pages containing memory-mapped I/O devices that have side effects or that are sensitive to order (such as queues).

The NaTPage attribute precludes nonspeculative loads to a page so marked and assures that all speculative loads to the page return a deferred-exception indicator to the destination register. IA-64 instruction fetches, stores, and nonspeculative loads, as well as all IA-32 memory accesses to a NaTPage, invoke a NaT-page-consumption fault. The purpose of the NaTPage is to enable aggressive use of speculative loads while avoiding constant reentry to the OS via exception because of speculative accesses to unmapped memory.

Relax for Better Performance

For cachable memory, IA-64 specifies a relaxed ordering model. In a relaxed model, most loads and stores have unordered semantics, allowing the hardware to rearrange them into an order different than that specified by the program, to improve performance. Memory-data dependencies impose order only between accesses to the same physical address and only from the same processor. To enforce order between memory operations to different addresses, or between different processors to any address, ordered semantics must be used.

For situations in which precise control over order is required—such as updating code images or sharing memory and synchronizing processes in a multiprocessor system—IA-64 provides strongly “ordered” memory-referencing instructions. These instructions impose an ordering relationship with respect to other orderable memory operations, according to one of three semantics: acquire, release, or fence.

Acquire semantics impose the restriction that the result of a memory instruction must be architecturally visible to all subsequent memory instructions. Release semantics require

the result of a memory instruction to be visible after the results of all previous memory instructions have been made visible. Fence semantics combine acquire and release semantics. In IA-64, ordered loads (*ld.acq*) have acquire semantics, ordered stores (*st.rel*) have release semantics, and a special memory-fence instruction (*mf*) enforces fence semantics. Acquire, release, and fence semantics are observed by all processors within the coherence domain.

In some multiprocessor situations, such as setting a lock on a shared data structure, even strict ordering isn't sufficient to assure foolproof operation. For this task, IA-64 provides three ordered semaphore instructions: exchange (*xchg*), compare and exchange (*cmpxchg*), and fetch and add (*fetchadd*). The *xchg* instruction always has acquire semantics, whereas the other two instructions have versions with either acquire or release semantics.

Most RISC processors—including MIPS, Alpha, and PowerPC—implement semaphores using a load-lock/store-conditional mechanism. This mechanism detects an interceding store to the semaphore location and requires software to retry the load-lock/store-conditional sequence until it completes atomically. IA-64, however, reverts to an approach used by some older processors, wherein special semaphore instructions perform a read, a modify, and a write operation as a single atomic (indivisible) operation.

Unlike some of the older processors that used bus locking in noncachable memory to make these operations indivisible, however, IA-64's semaphore instructions operate in cachable memory, as do load-lock/store-conditional instructions. Both systems rely on the cache-coherence mechanisms to implement indivisibility. But the load-lock/store-conditional mechanism has two drawbacks: first, it is difficult to guarantee forward progress (deadlock-free code), and second, it is not easy to assure that the semantics will remain consistent across multiple processor generations without placing serious restrictions on how these instructions are used. Because of the difficulties, and because of the fact that the load-lock/store-conditional mechanism is normally used to simulate compare-and-exchange or fetch-and-add primitives anyway, Huck says it was just easier, especially on the software trying to use them, to implement the full primitives directly in hardware.

Attribute	TLB.ma	Cachable	Coalescing	Sequential	Speculative	Coherent
Write back	000	Yes	No	No	Yes	Yes
Write Coalescing	110	No	Yes	No	Yes	No
Uncachable	100	No	No	Yes	No	Yes
Uncachable Exported	101	No	No	Yes	No	Yes
Reserved for Software	001	n/a	n/a	n/a	n/a	n/a
Reserved	010	n/a	n/a	n/a	n/a	n/a
Reserved	011	n/a	n/a	n/a	n/a	n/a
NaTPage	111	Yes	n/a	No	Yes	n/a

Table 2. Each virtual page can be assigned certain attributes that control whether data and instructions on the page are eligible to be cached, coalesced into groups for more efficient transfer to memory, performed out of order, performed speculatively, or maintained coherent by the processor/memory system. n/a = not applicable.

Wait Just a Millisecond

IA-64 processors implement precise interruptions: that is, following an interruption, the processor is put into a state that is consistent with in-order, non-pipelined execution of the program. Processor hardware is responsible for unwinding the pipeline, for saving the necessary state in interruption registers, and for vectoring control to a software interruption handler. After handling an interruption, software executes a return-

from-interruption (rfi) instruction, which restores the processor state and resumes execution of the interrupted program.

IA-64 interruptions fall into two broad categories: IVA-based and PAL-based. PAL-based interruptions are vectored through hardware entry points and are handled by the processor-abstraction-layer (PAL) firmware (described later), which runs beneath the level of the OS kernel. IVA-based interruptions are vectored through an interruption-vector table (IVT), as Table 3 shows, and are handled by the OS.

IA-64 provides a large number of very specific interruption vectors and captures a considerable amount of information about each interruption in control registers. These features improve interruption response time and latency by minimizing the amount of self-discovery that interruption handlers must go through to determine the cause of the interruption and to decide the correct course of action.

Vector Offset	Vector Name
0x0000	VHPT translation vector
0x0400	Instruction TLB vector
0x0800	Data TLB vector
0x0C00	Alternate instruction TLB vector
0x1000	Alternate data TLB vector
0x1400	Data nested TLB vector
0x1800	Instruction key miss vector
0x1C00	Data key miss vector
0x2000	Dirty-bit vector
0x2400	Instruction access-bit vector
0x2800	Data access-bit vector
0x2C00	Break instruction vector
0x3000	External interrupt vector
0x3400–0x4C00	Reserved
0x5000	Page not present vector
0x5100	Key permission vector
0x5200	Instruction access rights vector
0x5300	Data access rights vector
0x5400	General exception vector
0x5500	Disabled FP-register vector
0x5600	NaT-consumption vector
0x5700	Speculation vector
0x5800	Reserved
0x5900	Debug vector
0x5A00	Unaligned reference vector
0x5B00	Unsupported data-reference vector
0x5C00	Floating-point fault vector
0x5D00	Floating-point trap vector
0x5E00	Lower-privilege transfer trap vector
0x5F00	Taken-branch trap vector
0x6000	Single-step trap vector
0x6100–0x6800	Reserved
0x6900	IA-32 exception vector
0x6A00	IA-32 intercept vector
0x6B00	IA-32 interrupt vector
0x6C00–0x7100	Reserved

Table 3. Following an IVA-based interruption, control is transferred to the appropriate interruption-vector address (offset by the base address of the vector table in the IVT control register). A large number of interruption vectors are provided to minimize the work software handlers must do to determine the cause of an interruption.

There are four types of interruptions: aborts, interrupts, faults, and traps. (Note that in IA-64 terminology, “interruption” refers to all four types generically, whereas “interrupt” is one of the four specific types.) Aborts are PAL-based interruptions and include machine checks (MCAs), which occur in response to hardware errors, and processor reset (RESET). Aborts are the only type of interruptions that are not necessarily precise.

Interrupts include initialization interrupts (INITs), platform-management interrupts (PMIs), and external interrupts (INTs). INITs and PMIs are PAL-based interrupts, while INTs are IVA based. External interrupts typically come from I/O devices but can be generated by any processor in the system, including the one being interrupted.

In IA-64 systems, external interrupts are delivered as in-band-coded messages across the IA-64 system bus, as Figure 4 shows. This technique is more flexible and much easier to configure than the traditional method of interrupt delivery, which requires separate wires from all possible interrupt sources to all possible interrupt sinks. External interrupts are all delivered through IVT vector 0x3000, and the specific interrupt is read by the interrupt-handler software from the IVR control register. These interrupts are prioritized and vectored through 256 unique vectors, including one for a non-maskable interrupt (NMI) and another (ExtINT) for backward compatibility with venerable 8259A interrupt controllers (8259A interrupts are delivered through two discrete interrupt-input pins, LINT₀ and LINT₁).

The remaining two interruption types, faults and traps, occur in response to program-generated events; both are IVA based. Faults are generated as a result of some unintended or illegal operation, such as a page fault, division by zero, or an undefined opcode. Faults are synchronous

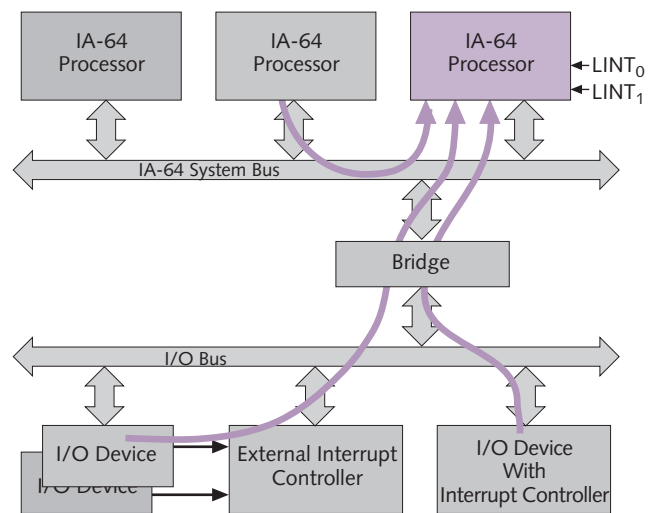


Figure 4. IA-64 systems deliver external interrupts (purple) as messages passed over the system bus, which greatly simplifies system configuration. Two traditional interrupt pins (LINT₀ and LINT₁) are provided for backward compatibility with 8259A interrupt controllers.

with instruction execution, and the fault handler is invoked after the previous instruction is complete but before the faulting instruction begins (or so it is made to appear to software). Traps occur in response to calls for operating system services. Like faults, traps are synchronous with the instruction stream, but they are invoked after the trapping instruction completes and before the next instruction begins.

Register Banks Reduce Interruption Latency

In most processors, a taken interrupt disables further interrupts until the interrupt handler has had a chance to safely tuck away the contents of the registers it needs onto the memory stack. But saving registers adds overhead to the handler itself, and it increases the worst-case latency to subsequent (nested) interrupts.

To reduce latency and speed low-level interruption handlers—such as those for TLB misses, updating a TLB-entry dirty bit, performing a misaligned memory access, or emulating an instruction—IA-64 provides an alternate register bank to hold OS data. The alternate register bank consists of 16 registers that, on a first-level interruption, are automatically switched into the general-register namespace as GR₁₆ through GR₃₁. A bit in the PSR (PSR.bn) controls which register bank is currently active, allowing the interrupt handler to quickly switch back to the application's registers if they are needed.

On a first-level interruption, the stack frame is not automatically changed, but the register-stack engine may spill or fill registers as the interruption handler runs. Because of the alternate register bank, however, the stack frame is not normally modified by low-level handlers, preventing high-speed handlers from having to manipulate the stack.

Huck says that the implementation cost of the extra register bank is minimal, because the extra register cells can be buried beneath the wires of the main register file. Perhaps this ability is an example of the benefits of EPIC design: in Athlon (see *MPR 8/23/99-01*, "Athlon Outruns Pentium III"), for example, the future file (reorder buffer) occupies the extra space under the register-file wires. Without a reorder buffer, however, Itanium can offer the extra-bank feature with no increase in die area.

Abstraction Layers Isolate Hardware

In an effort to create a common IA-64 environment capable of running shrink-wrapped operating systems on a variety of platforms, HP and Intel have defined a firmware environment that abstracts the platform and processor hardware. This abstraction layer makes low-level differences between processors and platforms invisible to low-level system and diagnostic software.

The IA-64 firmware environment consists of three major components, as Figure 5 shows. The extensible firmware interface (EFI) replaces some of the functions of the traditional BIOS and provides a procedure-call interface to the operating system for boot-time services, such as boot prompt, console, serial interface, and (eventually) network-boot facilities. The system abstraction layer (SAL) isolates the operating system from underlying differences in the platform hardware. And the processor abstraction layer (PAL), which is completely specified by the IA-64-processor architecture, serves to give the SAL and the OS a common view of different processor implementations.

Primarily, the PAL abstracts processor functions that are not performance critical and are not visible to application software, such as processor initialization, configuration, and error handling. Together, the PAL, SAL, and EFI layers are responsible for booting the operating system. For the most part, these layers play no part in running application code, although they do provide some run-time services to the OS for such things as machine checks and system resets.

Many of the processor resources that the PAL abstracts are accessed through architecturally defined or implementation-specific control registers by privileged code. While such control registers are present in most processors, they always create some implementation challenges. For one, it is often difficult to ensure that all the side effects of a control-register change are complete before beginning execution of any subsequent instruction that depends on the new machine context.

PA-RISC solved this problem with a rather arbitrary eight-instruction rule; that is, the effects had to be visible to any instruction eight or more instructions downstream from

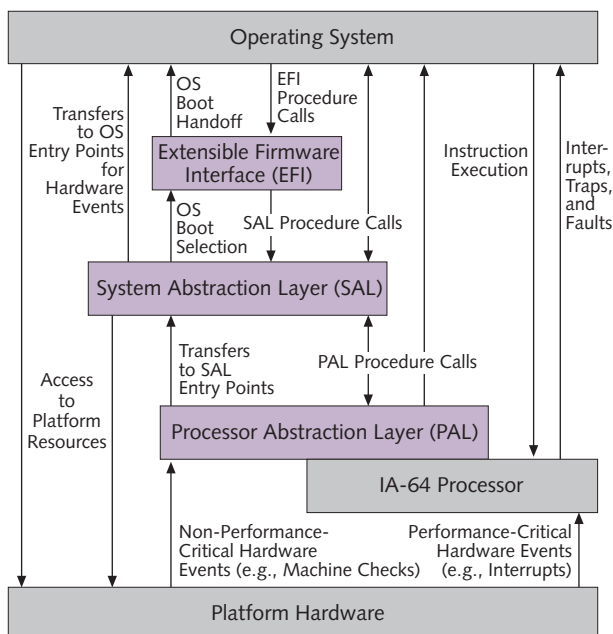


Figure 5. IA-64 architects envision a three-layer hardware-abstraction architecture to allow shrink-wrapped operating systems to run on multiple platforms. The processor abstraction layer makes all IA-64 processors appear identical to the OS—at least for features that are not critical to performance and that are likely to change from implementation to implementation.

the one that altered the control register. Such a rule, however, can be ambiguous and difficult to implement in a pipelined processor. To minimize these problems, IA-64 took an approach similar to that taken by PowerPC, which requires explicit serialization instructions to be inserted in the code stream at the point beyond which the effects of all pending control-register changes must be visible. For this purpose, IA-64 provides separate instructions for serializing control-register changes that affect instruction processing (serialize.i) and data processing (serialize.d).

End of an Era


The IA-64 architecture has been under development for more than a decade, beginning with early VLIW research at HP in 1989. Since 1994, HP and Intel have been working together intensely to define the architecture. And for more than two years now, the companies have been performing a carefully orchestrated strip-tease designed to hype the architecture, to build momentum, and to mark time while putting on the finishing touches. Now, with the system-architecture disclosures described in this article, the long disclosure process has, thankfully, come to a close.

Due to IA-64's radically different VLIW (okay, EPIC) instruction-set architecture, it may take several more years to fully develop the requisite compiler technology, to learn to craft high-performance chips around the new architecture, and to establish a software base large enough to enable IA-64 chips to penetrate the high-volume market. Although

For More Information

The IA-64 system architecture specification is available in HTML on HP's Web site at <http://devresource.hp.com/devresource/Docs/Refs/IA64ISA/index.html>. A more printer friendly Acrobat version is available on the Intel site at <http://developer.intel.com/design/ia-64/manuals/>.

the IA-64 system architecture is, for some, disappointingly conventional compared with the instruction-set architecture, that choice was necessary to give IA-64 its best chance of gaining a foothold in the market.

Piling radically new system architecture on top of an unconventional ISA would have required operating systems to be redesigned from the bottom up. It is unlikely that initial IA-64 chips (i.e., Itanium) will have sufficiently compelling performance to motivate OS vendors to undertake such a mammoth task. But this will not be necessary, thanks to IA-64's use of precise exceptions and sequential semantics, its support for both MAS and SAS with graceful extension to new features, and its convenient processor-abstraction layer. With this conservative approach to system architecture, HP and Intel have virtually guaranteed that a variety of operating systems will be available for IA-64 chips as soon as they are ready for prime time later this year. 

To subscribe to Microprocessor Report, phone 408.328.3900 or visit www.MDRonline.com