Itanium 2 Processor Microarchitecture

THE ITANIUM 2 PROCESSOR EXTENDS THE PROCESSING POWER OF THE ITANIUM PROCESSOR FAMILY WITH A CAPABLE AND BALANCED MICROARCHITECTURE. EXECUTING UP TO SIX INSTRUCTIONS AT A TIME, IT PROVIDES BOTH PERFORMANCE AND BINARY COMPATIBILITY FOR ITANIUM-BASED APPLICATIONS AND OPERATING SYSTEMS.

Cameron McNairy
Intel

Don Soltis
Hewlett-Packard

On 8 July 2002, Intel introduced the Itanium 2 processor—the Itanium architecture's second implementation. This event was a milestone in the cooperation between Intel and Hewlett-Packard to establish the Itanium architecture as a key workstation, server, and supercomputer building block. The Itanium 2 processor may appear similar to the Itanium processor, yet it represents significant advances in performance and scalability. (Sharangpani and Arora give an overview of the Itanium processor.[1]) These advances result from improvements in frequency, pipeline depth, pipeline control, branch prediction, cache design, and system interface. The microarchitecture design enables the processor to effectively address a wide variety of computation needs.

Table 1 lists the processor's main features. We obtained the Spec FP2000 and Spec CPU2000 benchmark results from http://www.spec.org on 20 February 2002. We obtained the other benchmarks from http://developer.intel.com/products/server/processors/server/itanium2/index.htm. This site contains relevant information about the measurement circumstances.

Microarchitecture overview

Many aspects of the Itanium 2 processor microarchitecture result from opportunities and requirements associated with Intel's Itanium architecture (formerly called the IA-64 architecture).[2] The architecture goes beyond simply defining 64-bit operations and register widths; it defines flexible memory management schemes and several tools that compilers can use to realize performance. It enables parallel instruction execution without resorting to complex out-of-order pipeline designs by explicitly indicating which instructions can issue in parallel without data hazards. To that end, three instructions are statically grouped into 16-byte bundles. Multiple instruction bundles can execute in parallel, or explicit stops can break parallel execution to avoid data hazards. Each bundle encodes a template that indicates which type of execution resource the instructions require: integer (I), memory (M), floating point (F), branch (B), and long extended (LX). Thus, memory, floating-point, and branch operations that can execute in parallel comprise a bundle with an MFB template.

The Itanium 2 processor designers took advantage of explicit parallelism to design an in-order, six-instruction-issue, parallel-execution pipeline. The relatively simple pipeline allowed the design team to focus resources on the memory subsystem's performance and to exploit many of the architecture's performance opportunities. Figure 1 shows the

Published by the IEEE Computer Society

0272-1732/03/$17.00 © 2003 IEEE

core pipeline and the relationship of some microarchitecture structures to the pipeline. These structures include the instruction buffer, which decouples the front end, where instruction fetch and branch prediction occur, from the back end, where instructions are dispersed and executed. The back-end pipeline renames virtual registers to physical registers, accesses the register files, executes the operation, checks for exceptions, and commits the results.

Instruction fetch

The front-end structures fetch instructions for later use by the back end. The front end chooses an instruction pointer (IP) from the next linear IP, branch prediction resteer pointers, or branch misprediction and instruction exception resteer pointers. The front end then presents the IP to the instruction cache and translation look-aside buffer (TLB). These structures are tightly coupled, allowing the processor to determine which cache way, if any, was a hit, and to deliver the cache contents in the next cycle using an innovation called prevalidated tags. This is the same idea presented in other Itanium 2 processor descriptions[3] in the context of the first-level data (L1D) cache, but here we discuss it in the context of the instruction cache.

Prevalidated-tag cache design

Traditional physically addressed cache designs require a TLB access to translate a virtual address to a physical address. The cache's hit detection logic then compares the physical address with the tags stored in each cache way. The serialized translation and comparison typically lead to multicycle cache designs. In a prevalidated-tag cache design, the cache tags do not store a physical address; they store an association to the TLB entry that holds the appropriate virtual-address translation. In the Itanium 2 processor, when the front end presents a virtual address to the TLB, the cache's detection logic directly compares the identifier of the entry that matches the virtual address, called the match line, with a one-hot vector stored in the cache tags. The vector indicates which TLB entry holds the translation associated with the contents of that cache way. This allows a fast determination of which cache way of a set, if any, is a hit. The hit result feeds into the way select logic to drive the cache contents to the consumer.

Table 1. Features of the Itanium 2 processor.

Design		
Frequency	1 GHz	
Pipe stages	8 in-order	
Issue/retire	6 instructions	
Execution units	2 integer, 4 memory, 3 branch, 2 floating-point	
Silicon		
Technology	180 nm	
Core	40 million transistors	
L3 cache	180 million transistors	
Size	421 mm^2	
Caches		
L1 instruction	Size	16 Kbytes
	Latency	1 cycle
	Protection	Parity
L1 data	Size	16 Kbytes
	Latency	1 cycle
	Protection	Parity
L2	Size	256 Kbytes
	Latency	5, 7, or 9+ cycles
	Protection	Parity or ECC*
L3	Size	3 Mbytes
	Latency	12+ cycles
	Protection	ECC
Benchmark results		
Spec CPU2000 score	810	
Spec FP2000 score	1,431	
TPCC (32-way)	433,107 transactions per minute	
Stream	3,700 Gbytes/s	
Linpack 10K**	13.94 Gflops	

* ECC: error-correcting code

** Performed with four processors

The removal of the physical address from the hit detection critical path is significant. It provides an opportunity for a single-cycle cache, but requires the TLB to be tightly coupled with the cache tags. Another implication is that a miss in the TLB also results in a cache miss, because no match lines will be driven. Moreover, the number of TLB entries determines the number of bits held in each way's tag and might limit the coupled TLB's size. Figure 2 shows how prevalidated tags tied to a 32-entry TLB determine a hit.

L1I cache complex

The L1I cache complex comprises the first-level instruction TLB (L1I TLB), the second-level instruction TLB (L2I TLB), and the first-level instruction cache (L1I). The L1I

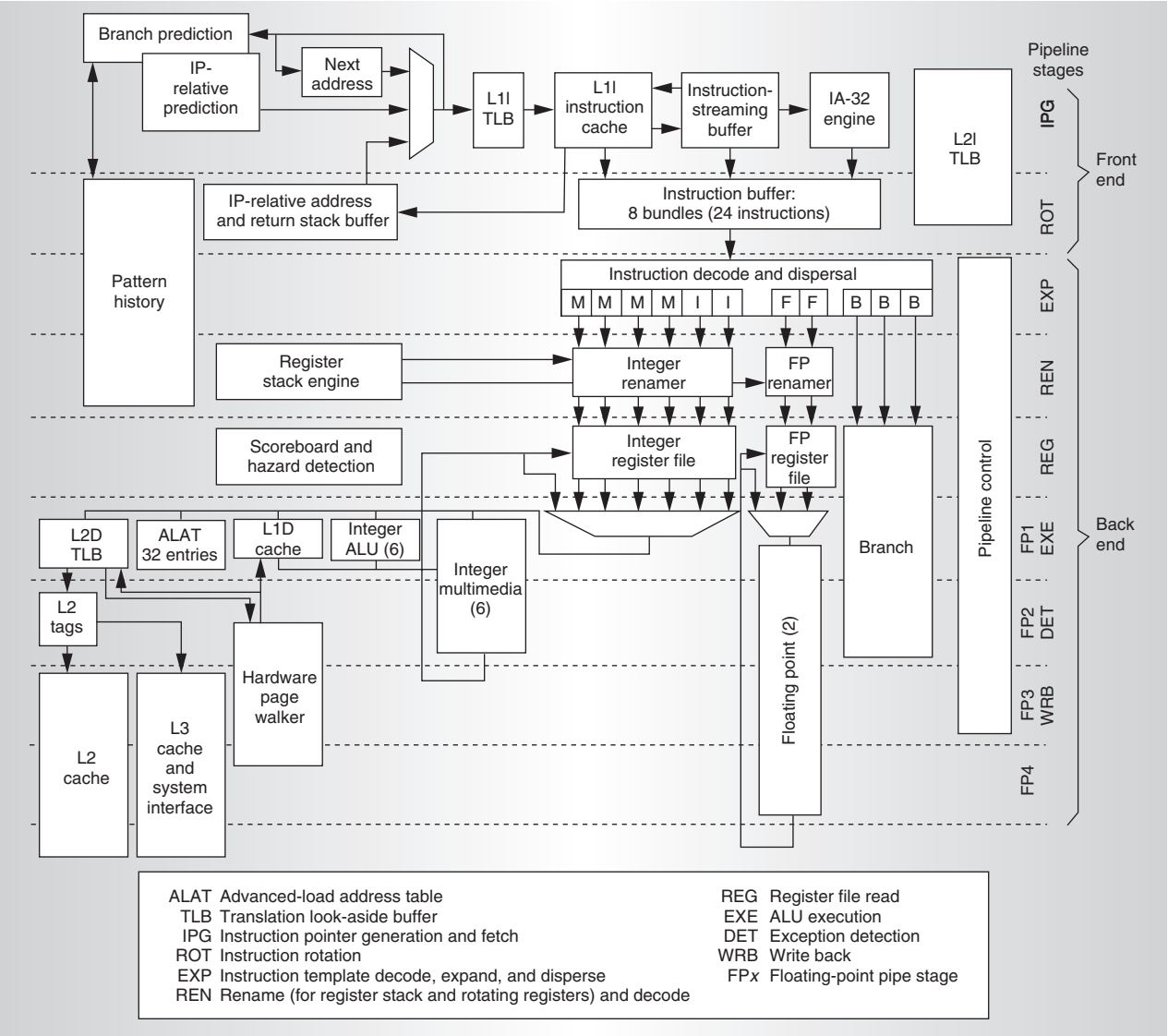

Figure 1. Itanium 2 processor pipeline.

TLB and the L1I cache are arranged as required for a prevalidated-tag design. The four-way set-associative L1I cache is 16 Kbytes in size, relatively small because of latency and area design constraints but still optimal. An instruction prefetch engine enhances the cache's effective size. The dual-ported tags and TLB resolve demand and prefetch requests without conflict. The page offset of the virtual-address bits selects a set from the tag array and the data array for demand accesses. The upper bits of the virtual address determine which, if any, way holds the requested instructions. The tag and TLB lookup results determine a L1I hit or miss, as described earlier.

The 64-byte L1I cache line holds four instruction bundles. The L1I can sustain a stream of one 32-byte read per cycle to provide two bundles per cycle to the back-end pipeline. The fetched bundles go directly to the dispersal logic or into an instruction buffer for later consumption. If the instruction buffer is full, the front-end pipeline stalls.

The L1I TLB directly supports only a 4-Kbyte page size. The L1I TLB indirectly supports larger page sizes by allocating additional entries as each 4-Kbyte segment of the larger page is referenced. An L1I TLB miss implies a miss in the L1I cache and can initiate L2I TLB and second-level (L2) cache accesses, as well as

a transfer of page information to the L1I TLB.

The L2I TLB is a 128-entry, fully associative structure with a single port. Each entry can represent all page sizes defined in the architecture from 4 Kbytes to 4 Gbytes. Up to 64 entries can be pinned as translation registers to ensure that hot pages are always available. In the event of an L2I TLB miss, the L2I TLB requests the hardware page walker (HPW) to fetch a translation from the virtual hashed page table. If a translation is available, the HPW inserts it into the L2I TLB. If a translation is not available or the HPW aborts, an exception occurs and the operating system assumes control to establish a mapping for the reference.

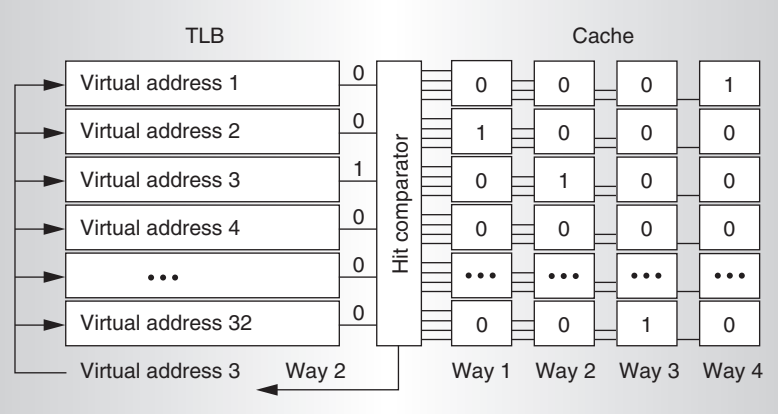

Figure 2. Prevalidated cache tags tied to the TLB determine a hit. The presented virtual address is TLB entry 3. The TLB drives a match line indicating the match to the hit comparator, which reads and compares the way's tags against this match line. The tag in way 2 matches the match line, so way 2 is reported as a hit.

Instruction-streaming buffer

The instruction-streaming buffer augments the instruction cache. The ISB holds eight L1I cache lines of instructions returned from the L2 or higher cache levels. It also stores virtual addresses that are scanned by the ISB hit detection logic for each IP presented to the L1I cache. An ISB hit has the same one-cycle latency as a normal L1I cache hit. Instructions typically spend little time in the ISB because the L1I cache can usually support reads and fills in the same cycle. The ISB enables branch prediction, instruction demand accesses, and instruction prefetch accesses to occur without conflict.

Instruction prefetching

Software can engage the instruction prefetch engine to reduce the instruction cache miss count and the associated penalty. The architecture defines hint instructions that provide the hardware early information about a future branch. In the Itanium 2 processor, these instructions direct the instruction prefetch engine to prefetch one or many L2 cache lines. The virtual address of the desired instructions allocates into the eight-entry prefetch virtual address buffer. Addresses from this buffer access the L1I TLB and L1I cache tags through the prefetch port, keeping prefetch requests from interfering with critical instruction access. If the instructions already exist in the L1I cache, the address is removed from the address buffer. If the instructions are missing, the prefetch engine sends a prefetch request to the L2 cache.

The prefetch engine also supports a special prefetch hint on branch instructions to initiate a streaming prefetch. For these hints, the prefetch engine continues to fetch along a linear path, up to four L2 cache lines ahead of demand accesses. Software hints can explicitly stop the current streaming prefetch or engage a new streaming prefetch. The prefetch engine automatically stops prefetching down a path if a mispredicted branch resteers the front end. The prefetch engine avoids cache pollution through software hints, branch-prediction-based cancellation, self-throttle mechanisms, and an L1I cache line replacement algorithm that biases unreferenced instructions for replacement.

Branch prediction

The Itanium 2 processor's branch prediction performance relies on a two-level prediction algorithm and two levels of branch history storage. The first level of branch prediction storage is tightly coupled to the L1I cache. This coupling allows a branch's taken/not taken history and a predicted target to be delivered with every L1I demand access in one cycle. The branch prediction logic uses the history to access a pattern history table and determine a branch's final taken/not taken prediction, or trigger, according to the Yeh-Patt algorithm.[4] The L2 branch cache saves the histories and triggers of branches evicted from the L1I so that they are available when the branch is revisited, providing the second storage level.

Table 2. Possible branch prediction penalties and their causes. A correctly predicted taken branch incurs no penalty.

Penalty (cycles)	Cause
1	Correctly predicted taken IP-relative branch with incorrect target and return branch
2	Nonreturn indirect branch
6	Incorrect taken/not taken prediction or incorrect indirect target

The one-cycle latency provides a zero-penalty resteer for correctly predicted IP-relative branches. The prediction information consists of the prediction history and trigger for every branch instruction, up to three per bundle, and a portion of the predicted target's virtual address for every bundle pair. Because the bundles share the target and the target may not be sufficient to represent the entire span required by the branch, there might be times when the front end is resteered to an incorrect address. The branch prediction logic tracks this situation and provides a corrected IP-relative target one cycle later.

Return stack buffer and indirect branches

All predictions for return branches come from an eight-entry return stack buffer. A branch call pushes both the caller's IP and its current function state onto the RSB. A return branch pops off this information. The RSB predictions resteer the front end two cycles after the cache lookup that contains the return branch.

The branch prediction logic predicts indirect branch targets on the basis of the current value in the referenced branch register three cycles after the cache lookup that contains the indirect branch.

Branch resolution

All branch predictions are validated in the back-end pipeline. The branch prediction logic allows in-flight branch prediction to determine future branch prediction behavior; however, nonspeculative prediction state is maintained and restored in the case of a misprediction. Table 2 lists the possible branch prediction penalties and their causes.

L2 branch cache

The size and organization of the branch information suggests that branch prediction accuracy suffers when the instruction stream revisits a branch that has lost its prediction history because of an eviction. To mitigate the potential loss of branch histories, the L2 branch cache stores the trigger and histories of branches evicted from the first-level storage. The L2B is a 24,000-entry backing store that does not use tags; instead it uses three address-based hashing functions and voting to determine the correct initialization of prediction histories and triggers for L1I fills. Limiting the L2B to prediction history and trigger but not target provides a highly effective and compact design. A branch target can be recalculated, in most cases, before a L1I fill occurs and with little penalty. It is possible that the L2B does not contain any information for the line being filled to L1I. In that case, the trigger and history bits are initialized according to the branch completers provided in the branch instruction.

Instruction buffer

The instruction buffer receives instructions from the L1I or L2 caches and lets the front end fetch instructions ahead of the back-end pipeline's consumption of instructions. This eight-bundle buffer and bundle rotator can present a wide combination of two-instruction bundles to back-end dispersal logic. Thus, no matter how many instructions the back end consumes in a cycle, two bundles of instructions are available. The dispersal logic indicates that zero, one, or two bundles were consumed so that the instruction buffer can free the appropriate entries. If the pipeline is flushed or the instruction buffer is empty, a bundle can bypass the instruction buffer completely.

Instruction dispersal

Figure 3 shows the design of the Itanium 2 processor front end and dispersal logic. The processor can issue and execute two instruction bundles, or six instructions, at a time. These instructions issue to one of 11 issue ports:

- two integer,
- four memory,
- two floating-point, and
- three branch.

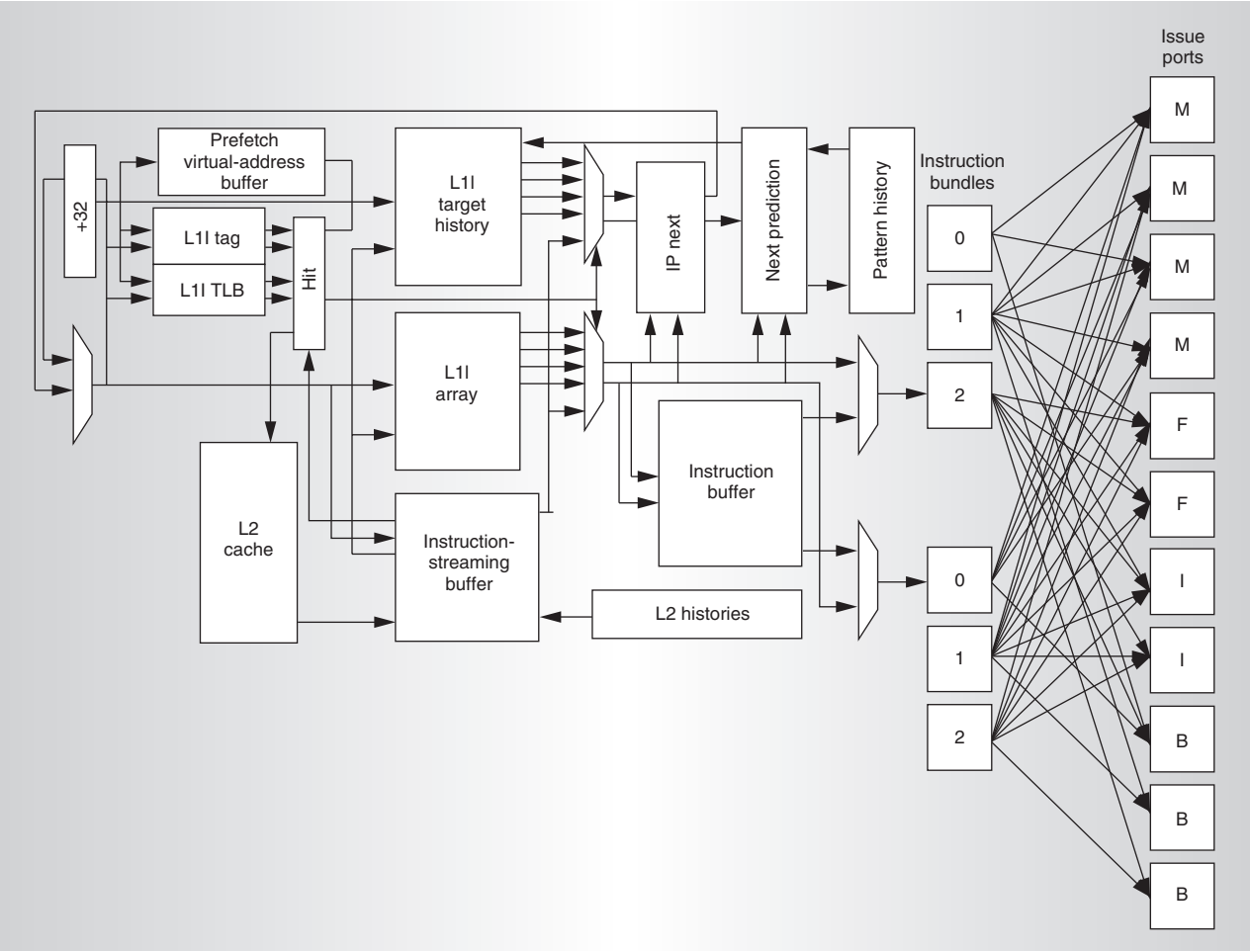

Figure 3. Itanium 2 processor front-end and dispersal-logic design.

These ports allocate instructions to several execution units. Two integer units execute integer operations such as shift and extract; ALU operations such as add, and, and compare; and multimedia ALU operations. Four memory units execute memory operations such as load, store, semaphore, and prefetch, in addition to the ALU and multimedia instructions that the integer units can execute. The four memory units are slightly asymmetric—two are dedicated to integer loads and two to stores. Compared with a two-memory-port implementation, the four memory ports provide a threefold increase in dual-issue template combinations and many other performance improvement opportunities.[5]

The processor's dispersal logic looks at two bundles of instructions every cycle and assigns as many instructions as possible to execution resources. There are multiple resources for each template type (I, M, F, B, and LX), and the dispersal logic typically assigns the first I instruction to the first I resource, the second I instruction to the second I resource, and so on until it exhausts the resources or an explicit stop bit breaks up an issue group. If instructions in the two bundles considered require more resources than available, the issue group stops at the oversubscription point, and the remaining instructions wait for dispersal in the next cycle. The instruction in an issue group is determined at dispersal and remains constant through the in-order execution pipeline.

The dispersal logic dynamically maps instructions to the most appropriate resource. This is important in cases of limited or asymmetric execution resources. For example, the dispersal logic assigns a load instruction to the first load-capable M port (M0 or M1) and a store to the first store-capable M port (M2 or

M3) even if the store precedes the load in the issue group. In addition, the dispersal logic ignores this asymmetry for floating-point loads so that they issue to any M resource. Dynamic resource mapping also lets instructions typically assigned to I resources issue on M resources. If the template assigns an ALU or multimedia operation to an I resource, but all I resources have been exhausted, the dispersal logic dynamically reassigns the operation to an available M resource. Thus, the processor can often issue a pair of MII bundles despite having only two I resources. These capabilities remove the burden of ordering and padding instructions to ensure that they issue to correct resources from the code generator.

Register stack engine and register renaming

The Itanium 2 processor implements 128 integer registers, a register stack engine (RSE), and register renaming. These features work together to give software the perception of unlimited registers. The RSE maintains the register set currently visible to the application by saving registers to and restoring registers from a backing store. Software can allocate registers as needed, and the RSE can stall the pipeline to write a dirty register's value to memory (backing store) to make room for the newly allocated registers. A branch return instruction can also engage the RSE if the registers required by the return frame are not available and must be loaded from the backing store. The RSE significantly reduces the number of memory operations required by software for function and system calls. The larger register file and the RSE implementation keep the amount of time applications spend saving and restoring registers low.[6]

The register-renaming logic manages registers across calls and returns and enables efficient software-pipelined code. The logic works with branch instructions to provide a new set of registers to a software-pipelined loop through register rotation. Compilers can software-pipeline many floating-point and a growing number of integer loops, resulting in significant code and execution efficiencies. In addition, the static nature of renaming, from RSE engagement onward, means that this capability is far simpler than the register renaming performed by out-of-order implementations.

Scoreboard and hazard detection

The Itanium 2 processor's scoreboard mechanism enables high performance in the face of L1D misses. Once physical register identifiers are available from the register-renaming logic, the hazard detection logic compares them with the register scoreboard, which lists registers associated with earlier L1D misses. If an instruction source or destination register matches a scoreboarded register, the issue group stalls at the execute (EXE) stage until the register value becomes available. This feature facilitates nonblocking-cache designs, in which multiple cache misses can be outstanding and yet the processor can continue to execute until the instruction stream references a register in the scoreboard.

A similar mechanism exists for other long-execution-latency operations such as floating-point and multimedia operations. For them, latency is fixed at four and two cycles, respectively. The hazard detection logic tracks the operation type and destination register and compares each instruction source and destination register with these in-flight operations. Like a scoreboard match, a long-latency-operation match stalls the entire issue group.

Integer execution and bypass

The six execution units supporting integer, multimedia, and ALU operations are fully bypassed; that is, as soon as an execution unit calculates a result, the result becomes available for use by another instruction on any other execution unit. A producer-and-consumer dependency matrix, considering latencies and instruction types, controls the bypass network. Twelve read ports and eight write ports on the integer register file and 20 bypass choices support highly parallel execution.[7] Six of the eight write ports are for calculation results, and the other two provide write paths for load returns from the L1D cache. All ALU and integer operations complete in one cycle.

Floating-point execution

Each of the two floating-point execution units can execute a fused multiply-add or a miscellaneous floating-point operation. Latency is fixed at four cycles for all floating-point calculations. The units are fully pipelined and bypassed. Eight read and six

write ports access the 128 floating-point registers. Six of the read ports supply operands for calculation; the remaining two read ports are for floating-point store operations. Two of the write ports are for calculation results; the other four provide write paths for floating-point load returns from the L2 cache. The four M resources and the two F resources combined allow two MMF bundles to execute every cycle. This provides the memory and computational bandwidth required for technical computing.[5]

Table 3. Potential pipeline stalls.

Stage	Cause of stall
Rename	RSE activity required
Execute	Scoreboard and hazard detection logic
Exception detect	L2D TLB miss; L2 cache resources unavailable for memory operations; floating-point and integer pipeline coordination to avoid possible floating-point traps; or L1D and integer pipeline coordination

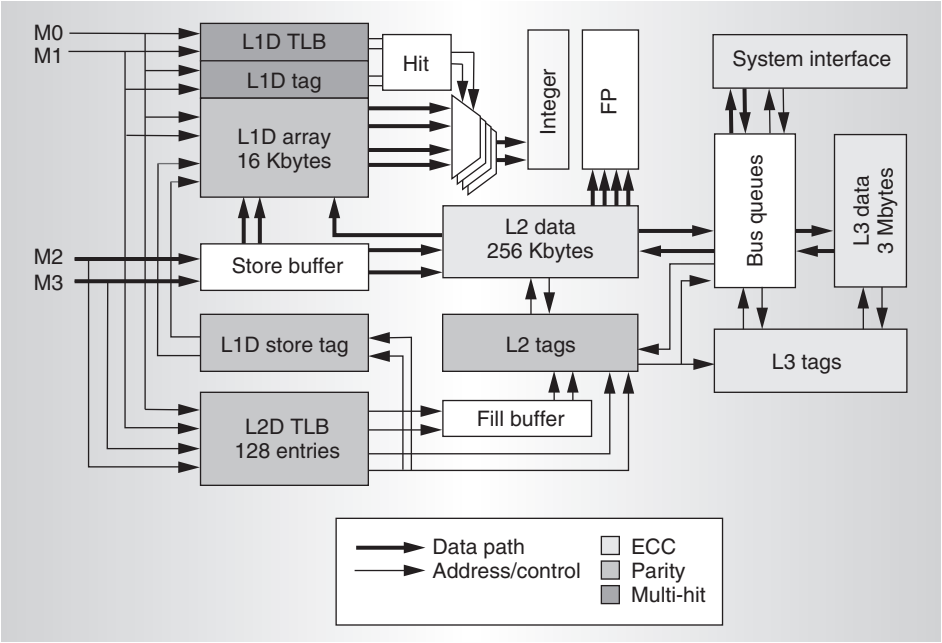

Figure 4. Itanium 2 processor's memory subsystem and system interface.

Pipeline control

The Itanium 2 processor pipeline is fully interlocked such that a stall in the exception detect (DET) stage propagates to the instruction expand (EXP) stage and suspends instruction advancement. A stall caused by one instruction in the issue group stalls the entire issue group and never causes the core pipeline to flush and replay. The DET-stage stall is the last opportunity for an instruction to halt execution before the pipeline control logic commits it to architectural state. The pipeline control logic also synchronizes the core pipeline and the L1D pipeline at the DET stage. The control logic allows these loosely coupled pipelines to lose synchronization so that the L1I and L2 caches can insert noncore requests into the memory pipeline with minimal impact on core instruction execution. Table 3 lists the stages and causes of potential stalls.

Memory subsystem

The relatively simple nature of the in-order core pipeline allowed the Itanium 2 processor designers to focus on the memory subsystem. The processor implements a full complement of region identifiers and protection keys, along with 64 bits of virtual address and 50 bits of physical address to provide 1,024 Tbytes of addressability. The memory subsystem is a low-latency, high-bandwidth design partitioned and organized to handle integer, floating-point, and enterprise workloads.[5] Figure 4 shows a simplified diagram of the memory subsystem and system interface, including some data and control paths and data integrity features.

Advanced-load address table

The advanced-load address table (ALAT) provides the ability to transform multicycle load accesses into zero-cycle accesses through dynamic memory disambiguation. The pipeline and scoreboard designs encourage scheduling loads as far ahead of use as possible to avoid core pipeline stalls. Unknown data dependencies normally prevent a compiler from scheduling a load much earlier than its use. The ALAT enables a load to advance beyond an unknown data dependency by

resolving that dependency dynamically. An advanced load allocates an entry in the ALAT, a four-ported, 32-entry, fully associative structure that records the register identifiers and physical addresses of advanced loads. A later store to the same address invalidates all overlapping ALAT entries. Later, when an instruction requires the load's result, the ALAT indicates whether the load is still valid. If so, a use of the load data is allowed in the same cycle as the check without penalty. If a valid entry is not found, the load is automatically reissued and the use is replayed. The Itanium architecture allows scheduling of a use in the same issue group as the check; hence, from the code scheduler's perspective, an ALAT hit has zero latency.

L1D cache

The data TLBs and L1D cache are similar in design to the instruction TLBs and caches; they share key attributes such as size, latency, arrangement, and tight integration with tags and the first-level TLB. The principle of prevalidated tags enables a one-cycle L1D cache. This feature is essential for a wide in-order microprocessor to achieve high performance in many integer workloads. If the latency were two cycles, the compiler would need to schedule at least five, and often more, instructions to cover the latency. The Itanium 2 processor's single-cycle latency requires only an explicit stop between a load and its use, thus easing the burden on the code generator to extract instruction-level parallelism.

The L1D is a multiported, 16-Kbyte, four-way set-associative, physically addressed cache with a 64-byte line protected by parity. Instructions access the L1D in program order; hence, it is an in-order cache. However, the scoreboard logic allows the L1D and other cache levels to be nonblocking. The L1D provides two dedicated load ports and two dedicated store ports. These ports are fixed, but the dispersal logic rearranges loads and stores within an issue group to ensure they reach the appropriate memory resource. The two load requests can hit and return data from the L1D in parallel without conflict. Rotators between the data array and the register file allow integer loads to any unaligned data reference within an 8-byte datum, as well as support for big- or little-endian accesses.

The prevalidated tags and first-level TLB serve only integer loads. Stores access the second-level data (L2D) TLB and use a traditional tagging mechanism. This increases their latency, but store latency is not a performance issue, in part because store-load forwarding is provided in the store data path. The L1D enforces a write-through with a no-write-allocate policy such that it passes all stores to the L2 cache, and store misses do not allocate into the L1D. If a store hits in the L1D, the data moves to a store buffer until the data array becomes available to update the L1D. These store buffers can merge store data from other stores and forward their contents to later loads.

Integer load and data prefetch misses allocate into the L1D, according to temporal hints and available resources. Up to eight L1D lines can have fill requests outstanding, but the total number of permitted L1D misses is limited only by the scoreboard and the other cache levels. If the L2 cannot accept a request, it applies back pressure and the core pipeline stalls. Before an L1D load miss or store request is dispatched to the L2, it accesses the L2D TLB. The TLB access behavior for loads differs from that of the instruction cache: The L1D and L2D TLBs are accessed in parallel for loads, regardless of an L1D hit or miss. This reduces both L1D and L2 latency. Consequently, the 128-entry, fully associative L2D TLB is fully four-ported to allow the complete issue of every possible combination of four memory operations.

The L1D is highly integrated into the integer data path and the L2 tags. All integer loads must go through the L1D to return data to the register file and core bypass network. The L1D pipeline processes all memory accesses and requests that need access to the L2 tags or the integer register file. Accordingly, several types of requests arbitrate for access to the L1D. Some of these requests have higher priority than core requests, and if there are conflicts, the core memory request stalls the core and reissues to the L1D when resources are available.

L2 cache

The second-level (L2) cache is a unified, 256-Kbyte, eight-way set-associative cache with a 128-byte line size. The L2 tags are true four-ported, with tag and ownership state protected by parity. The tags, accessed as part of the L1D pipeline, provide an early L2 hit or

miss indication. The L2 enforces write-back and write-allocate policies. The L2's integer access latency is five, seven, nine, or more cycles. Floating-point accesses require an additional cycle for converting to the floating-point register format.

The L2 cache is nonblocking and out of order. All memory operations that access the L2 (L1D misses and all stores) check the L2 tags and are allocated to a 32-entry queuing structure called the L2 OzQ. All stores require one of the 24 L2 data entries to hold the store until the L2 data array is updated. L1I instruction misses also go to the L2 but are stored in the instruction fetch FIFO (IFF) queue. Requests in the L2 OzQ and the IFF queue arbitrate for access to the data array or the L3 cache and system interface. This arbitration depends on the type of IFF request; instruction demand requests issue before data requests, and data requests issue before instruction prefetch requests. Up to four L2 data operations and one request to the L3 and system interface can issue every cycle.

The L2 OzQ maintains all architectural ordering between memory operations, while allowing unordered accesses to complete out of order. This makes specifying a single L2 latency difficult but helps ensure that older memory operations do not impede the progress of younger ones. In many cases, incoming requests bypass allocation to the L2 OzQ and access the data array immediately. This provides the five-cycle latency mentioned earlier. Sometimes the request can bypass the OzQ, but an L2 resource conflict forces the request to have a seven-cycle latency. The minimum latency for a request that issues from the L2 OzQ is nine cycles. Resource conflicts, ordering requirements, or higher-priority operations can extend a request's latency beyond nine cycles.

The L2 data array has 16 banks; each bank is 16 bytes wide and ECC-protected. The array allows multiple simultaneous accesses, provided each access is to a different bank. Floating-point loads can bypass or issue from the L2 OzQ, access the L2 data array, complete four requests at a time, and fully utilize the L2's four data paths to the floating-point units and register file. The L2 does not have direct data paths to the integer units and register file; integer loads deliver data via the L1D, which has two data paths to the integer units and register file. Stores can bypass or issue from the L2 OzQ and access the L2 data array four at a time, provided they access different banks.

The fill path width from the L2 to the L1D and the L1I is 32 bytes, requiring two cycles to transfer a 64-byte L1I or L1D line. The fill bandwidth from the L3 or system interface to the L2 is also 32 bytes per cycle. Four 32-byte quantities accumulate in the L2 fill buffers for either the L3 or system interface, allowing the interleaving of system interface and L3 data returns. The 128-byte cache line is written into the L2 in one cycle, updating both tag and data arrays.

L3 cache and system interface

All cacheable requests that the L2 cannot satisfy arrive at the L3 and the system interface. The L2 can make partial line requests of the system interface for uncacheable and write-coalescing accesses, but all cacheable requests are 128-byte accesses. The L2 can make one request to the L3 and the system interface per cycle. Requests enter one of the 16 bus request queues (BRQs) maintained by the system interface control logic. The BRQ can then send each request to the L3 to determine whether the L3 can satisfy the request. To lower the L3 access latency, an L2 request can bypass the BRQ allocation and query the L3 immediately. If the request is an L3 miss, it is scheduled to access the system interface. When the system interface responds with the data, the line is written to the L2 and the L3 in accordance with its temporal locality hints and access type.

L3 cache. The third-level cache is a unified, 3-Mbyte, 12-way set-associative cache with a 128-byte line size. Its access latency can be as low as 12 cycles, largely because the entire cache is on chip. All L3 accesses return an entire 128-byte line; the L3 doesn't support partial line accesses. The single-ported L3 tag array has ECC protection and is pipelined to allow a new tag access every cycle. The L3 data array is also single-ported and has ECC protection, but it requires four cycles to transfer a full data line to the L2 cache or the system interface. Requests that fill the L3 require four cycles to transfer data, using a separate data

path so that L3 reads and writes can be pipelined for maximum bandwidth. The L3 is nonblocking and has an eight-entry queue to support multiple outstanding requests. This queue orders requests and prioritizes them among tag read or write and data read or write to achieve the highest performance.

System interface. The system interface operates at 200 MHz and includes multiple subbuses for various functions, such as address/request, snoop, response, data, and defer. All buses, except the snoop bus, are protected against errors by parity or ECC. The data bus is 128 bits wide and operates source-synchronously at 400 million data transfers, or 6.4 Gbytes, per second. The system interface seamlessly supports up to four Itanium 2 processors.

The system interface control logic contains an in-order queue (IOQ) and an out-of-order queue (OOQ), which track all transactions pending completion on the system interface. The IOQ tracks a request's in-order phases and is identical on all processors and the node controller. The OOQ holds only deferred processor requests. The IOQ can hold eight requests, and the OOQ can hold 18 requests. The system interface logic also contains two 128-byte coalescing buffers to support write-coalescing stores. The buffers can coalesce store requests at byte granularity, and they strive to generate full line writes for best performance. Writes of 1 to 8 bytes, 16 bytes, or 32 bytes are possible when holes exist in the coalescing buffers.

The similarities between the system interfaces of the Itanium 2 and Itanium processors allowed several implementations to leverage their Itanium-based solutions for use with the Itanium 2 processor. However, large, multinode system designs required additional support for high performance and reliability. As a result, the processors' system interface defines a few new transactions. The read current transaction lets the node controller obtain a current copy of data in a processor, while allowing the processor to maintain ownership of the line. The cache line replacement transaction informs a multinode snoop directory that an L3 clean eviction occurred to remove unnecessary snoop traffic. The cleanse cache transaction pushes a modified cache line out to system memory. This allows higher-performance processor check pointing in high-availability systems without forcing the processor to give up ownership of the line.

Soon after the Itanium 2 processor's introduction, major computer system providers either announced or introduced single- and dual-processor workstations, four- to 128-processor servers, and a 3,300-processor supercomputer, all using the Itanium 2 processor. The operating systems available for these systems include HPUX, Linux, and Windows .NET, and will eventually include OpenVMS. These systems and operating systems target a diverse set of computing problems and use the processor effectively for workstation, server, and supercomputer workloads. The Itanium 2 processor fits well in such varied environments because of its balanced design from instruction fetch to system interface and its flexible underlying architecture. The design team capitalized on the performance opportunities available in the Itanium architecture to produce a high-performance, in-order implementation and provide computer system developers a powerful and versatile building block. MICRO

References

1. H. Sharangpani and K. Arora, "Itanium Processor Microarchitecture," *IEEE Micro*, vol. 20, no. 5, Sept.-Oct. 2000, pp. 24-43.
2. J. Huck et al., "Introducing the IA-64 Architecture" *IEEE Micro*, vol. 20, no. 5, Sept.-Oct. 2000, pp. 12-23.
3. D. Bradley, P. Mahoney, and B. Stackhouse, "The 16KB Single-Cycle Read Access Cache on a Next Generation 64b Itanium Microprocessor," *Proc. 2002 IEEE Int'l Solid-State Circuits Conf.* (ISSCC 02), IEEE Press, 2002, pp. 110-111.
4. T.-Y. Yeh and Y.N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," *Proc. 19th Int'l Symp. Computer Architecture* (ISCA 92), ACM Press, 1992, pp. 124-134.
5. T. Lyon et al., "Data Cache Design Considerations for the Itanium 2 Processor," *Proc. 2002 IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors* (ICCD 02), IEEE Press, 2002, pp. 356-362.
6. J. McCormick and A. Knies, "A Brief

Analysis of the SPEC CPU2000 Benchmarks on the Intel Itanium 2 Processor," 2002; http://www.hotchips.org/archive/index.html.
7. E.S. Fetzer and J.T. Orton, "A Fully-Bypassed 6-Issue Integer Datapath and Register File on an Itanium Microprocessor," *Proc. 2002 IEEE Int'l Sold-State Circuits Conf.* (ISSCC 02), IEEE Press, 2002, pp. 420-478.

Cameron McNairy is an Itanium 2 processor microarchitect at Intel. His research interests include high-performance technical computing and large-system design issues. McNairy has a BSEE and an MSEE from Brigham Young University. He is a member of the IEEE.

Don Soltis is an Itanium 2 processor microarchitect at Hewlett-Packard. His research interests include microprocessor cache design and microprocessor verification. Soltis has a BSEE and an MSEE from Colorado State University.

Direct questions or comments about this article to Cameron McNairy, 3400 E. Harmony Road, MS 55, Fort Collins, CO 80526; cameron.mcnairy@intel.com.