ITANIUM PROCESSOR MICROARCHITECTURE

THE ITANIUM PROCESSOR EMPLOYS THE EPIC DESIGN STYLE TO EXPLOIT INSTRUCTION-LEVEL PARALLELISM. ITS HARDWARE AND SOFTWARE WORK IN CONCERT TO DELIVER HIGHER PERFORMANCE THROUGH A SIMPLER, MORE EFFICIENT DESIGN.

Harsh Sharangpani
Ken Arora
Intel

The Itanium processor is the first implementation of the IA-64 instruction set architecture (ISA). The design team optimized the processor to meet a wide range of requirements: high performance on Internet servers and workstations, support for 64-bit addressing, reliability for mission-critical applications, full IA-32 instruction set compatibility in hardware, and scalability across a range of operating systems and platforms.

The processor employs EPIC (explicitly parallel instruction computing) design concepts for a tighter coupling between hardware and software. In this design style the hardware-software interface lets the software exploit all available compilation time information and efficiently deliver this information to the hardware. It addresses several fundamental performance bottlenecks in modern computers, such as memory latency, memory address disambiguation, and control flow dependencies.

EPIC constructs provide powerful architectural semantics and enable the software to make global optimizations across a large scheduling scope, thereby exposing available instruction-level parallelism (ILP) to the hardware. The hardware takes advantage of this enhanced ILP, providing abundant execution resources. Additionally, it focuses on dynamic runtime optimizations to enable the compiled code schedule to flow through at high throughput. This strategy increases the synergy between hardware and software, and leads to higher overall performance.

The processor provides a six-wide and 10-stage deep pipeline, running at 800 MHz on a 0.18-micron process. This combines both abundant resources to exploit ILP and high frequency for minimizing the latency of each instruction. The resources consist of four integer units, four multimedia units, two load/store units, three branch units, two extended-precision floating-point units, and two additional single-precision floating-point units (FPUs). The hardware employs dynamic prefetch, branch prediction, nonblocking caches, and a register scoreboard to optimize for compilation time nondeterminism. Three levels of on-package cache minimize overall memory latency. This includes a 4-Mbyte level-3 (L3) cache, accessed at core speed, providing over 12 Gbytes/s of data bandwidth.

The system bus provides glueless multiprocessor support for up to four-processor systems and can be used as an effective building block for very large systems. The advanced FPU delivers over 3 Gflops of numeric capability (6 Gflops for single precision). The balanced core and memory subsystems provide

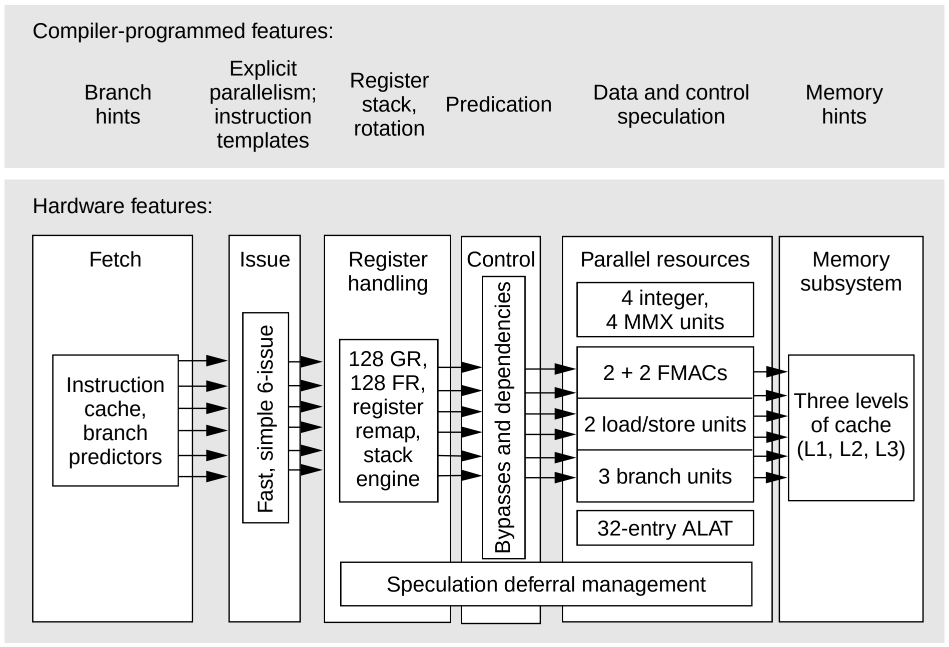

Compiler-programmed features:
Branch hints
Explicit parallelism; instruction templates
Register stack, rotation
Predication
Data and control speculation
Memory hints
Hardware features:
Fetch
Instruction cache, branch predictors
Issue
Fast, simple 6-issue
Register handling
128 GR, 128 FR, register remap, stack engine
Control
Bypasses and dependencies
Parallel resources
4 integer, 4 MMX units
2 + 2 FMACs
2 load/store units
3 branch units
32-entry ALAT
Memory subsystem
Three levels of cache (L1, L2, L3)
Speculation deferral management

Figure 1. Conceptual view of EPIC hardware. GR: general register file; FR: floating-point register file

high performance for a wide range of applications ranging from commercial workloads to high-performance technical computing.

In contrast to traditional processors, the machine's core is characterized by hardware support for the key ISA constructs that embody the EPIC design style.[1,2] This includes support for speculation, predication, explicit parallelism, register stacking and rotation, branch hints, and memory hints. In this article we describe the hardware support for these novel constructs, assuming a basic level of familiarity with the IA-64 architecture (see the "IA-64 Architecture Overview" article in this issue).

EPIC hardware

The Itanium processor introduces a number of unique microarchitectural features to support the EPIC design style.[2] These features focus on the following areas:

- supplying plentiful fast, parallel, and pipelined execution resources, exposed directly to the software;
- supporting the bookkeeping and control for new EPIC constructs such as predication and speculation; and
- providing dynamic support to handle events that are unpredictable at compilation time so that the compiled code flows through the pipeline at high throughput.

Figure 1 presents a conceptual view of the EPIC hardware. It illustrates how the various EPIC instruction set features map onto the micropipelines in the hardware.

The core of the machine is the wide execution engine, designed to provide the computational bandwidth needed by ILP-rich EPIC code that abounds in speculative and predicated operations.

The execution control is augmented with a bookkeeping structure called the advanced load address table (ALAT) to support data speculation and, with hardware, to manage the deferral of exceptions on speculative execution. The hardware control for speculation is quite simple: adding an extra bit to the data path supports deferred exception tokens. The controls for both the register scoreboard and bypass network are enhanced to accommodate predicated execution.

Operands are fed into this wide execution core from the 128-entry integer and floating-point register files. The register file addressing undergoes register remapping, in support of

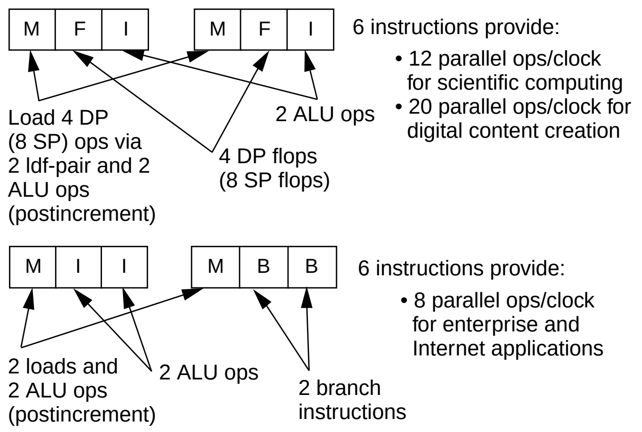

M F I M F I
6 instructions provide:
• 12 parallel ops/clock for scientific computing
• 20 parallel ops/clock for digital content creation
Load 4 DP (8 SP) ops via 2 ldf-pair and 2 ALU ops (postincrement)
4 DP flops (8 SP flops)
2 ALU ops
M I I M B B
6 instructions provide:
• 8 parallel ops/clock for enterprise and Internet applications
2 loads and 2 ALU ops (postincrement)
2 ALU ops
2 branch instructions

Figure 2. Two examples illustrating supported parallelism. SP: single precision, DP: double precision

register stacking and rotation. The register management hardware is enhanced with a control engine called the register stack engine that is responsible for saving and restoring registers that overflow or underflow the register stack.

An instruction dispersal network feeds the execution pipeline. This network uses explicit parallelism and instruction templates to efficiently issue fetched instructions onto the correct instruction ports, both eliminating complex dependency detection logic and streamlining the instruction routing network. A decoupled fetch engine exploits advanced prefetch and branch hints to ensure that the fetched instructions will come from the correct path and that they will arrive early enough to avoid cache miss penalties. Finally, memory locality hints are employed by the cache subsystem to improve the cache allocation and replacement policies, resulting in a better use of the three levels of on-package cache and all associated memory bandwidth.

EPIC features allow software to more effectively communicate high-level semantic information to the hardware, thereby eliminating redundant or inefficient hardware and leading to a more effective design. Notably absent from this machine are complex hardware structures seen in dynamically scheduled contemporary processors. Reservation stations, reorder buffers, and memory ordering buffers are all replaced by simpler hardware for speculation. Register alias tables used for register renaming are replaced with the simpler and semantically richer register-remapping hardware. Expensive register dependency-detection logic is eliminated via the explicit parallelism directives that are precomputed by the software.

Using EPIC constructs, the compiler optimizes the code schedule across a very large scope. This scope of optimization far exceeds the limited hardware window of a few hundred instructions seen on contemporary dynamically scheduled processors. The result is an EPIC machine in which the close collaboration of hardware and software enables high performance with a greater degree of overall efficiency.

Overview of the EPIC core

The engineering team designed the EPIC core of the Itanium processor to be a parallel, deep, and dynamic pipeline that enables ILP-rich compiled code to flow through at high throughput. At the highest level, three important directions characterize the core pipeline:

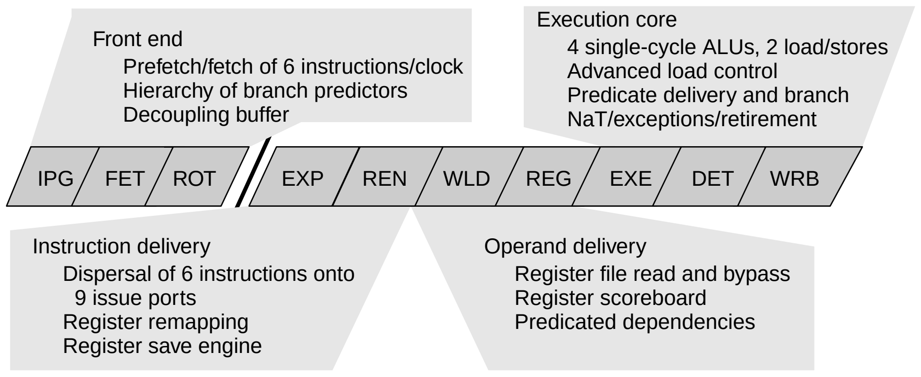
- wide EPIC hardware delivering a new level of parallelism (six instructions/clock),
- deep pipelining (10 stages) enabling high frequency of operation, and
- dynamic hardware for runtime optimization and handling of compilation time indeterminacies.

New level of parallel execution

The processor provides hardware for these execution units: four integer ALUs, four multimedia ALUs, two extended-precision floating-point units, two additional single-precision floating-point units, two load/store units, and three branch units. The machine can fetch, issue, execute, and retire six instructions each clock cycle. Given the powerful semantics of the IA-64 instructions, this expands to many more operations being executed each cycle. The "Machine resources per port" sidebar on p. 31 enumerates the full processor execution resources.

Figure 2 illustrates two examples demonstrating the level of parallel operation supported for various workloads. For enterprise and commercial codes, the MII/MBB template combination in a bundle pair provides six instructions or eight parallel operations per clock (two load/store, two general-purpose ALU operations, two postincrement ALU operations, and two branch instructions). Alternatively, an MIB/MIB pair allows the same mix of operations, but with one branch hint and one branch operation, instead of two branch operations.

For scientific code, the use of the MFI template in each bundle enables 12 parallel operations per clock (loading four double-precision operands to the registers and executing four double-precision floating-point, two integer ALU, and two postincrement ALU operations). For digital content creation codes that use single-precision floating point, the SIMD (single instruction, multiple data) features in the machine effectively enable up to 20 parallel operations per clock (loading eight single-precision operands, executing eight single-precision floating-point, two integer ALU, and two postincrementing ALU operations).



Front end
Prefetch/fetch of 6 instructions/clock
Hierarchy of branch predictors
Decoupling buffer
Execution core
4 single-cycle ALUs, 2 load/stores
Advanced load control
Predicate delivery and branch
NaT/exceptions/retirement
IPG
FET
ROT
EXP
REN
WLD
REG
EXE
DET
WRB
Instruction delivery
Dispersal of 6 instructions onto 9 issue ports
Register remapping
Register save engine
Operand delivery
Register file read and bypass
Register scoreboard
Predicated dependencies

Figure 3. Itanium processor core pipeline.

Deep pipelining (10 stages)

The cycle time and the core pipeline are balanced and optimized for the sequential execution in integer scalar codes, by minimizing the latency of the most frequent operations, thus reducing dead time in the overall computation. The high frequency (800 MHz) and careful pipelining enable independent operations to flow through this pipeline at high throughput, thus also optimizing vector numeric and multimedia computation. The cycle time accommodates the following key critical paths:

- single-cycle ALU, globally bypassed across four ALUs and two loads;
- two cycles of latency for load data returned from a dual-ported level-1 (L1) cache of 16 Kbytes; and
- scoreboard and dependency control to stall the machine on an unresolved register dependency.

The feature set complies with the high-frequency target and the degree of pipelining—aggressive branch prediction, and three levels of on-package cache. The pipeline design employs robust, scalable circuit design techniques. We consciously attempted to manage interconnect lengths and pipeline away secondary paths.

Figure 3 illustrates the 10-stage core pipeline. The bold line in the middle of the core pipeline indicates a point of decoupling in the pipeline. The pipeline accommodates the decoupling buffer in the ROT (instruction rotation) stage, dedicated register-remapping hardware in the REN (register rename) stage, and pipelined access of the large register file across the WLD (word line decode) and REG (register read) stages. The DET (exception detection) stage accommodates delayed branch execution as well as memory exception management and speculation support.

Dynamic hardware for runtime optimization

While the processor relies on the compiler to optimize the code schedule based upon the deterministic latencies, the processor provides special support to dynamically optimize for several compilation time indeterminacies. These dynamic features ensure that the compiled code flows through the pipeline at high throughput. To tolerate additional latency on data cache misses, the data caches are nonblocking, a register scoreboard enforces dependencies, and the machine stalls only on encountering the use of unavailable data.

We focused on reducing sensitivity to branch and fetch latency. The machine employs hardware and software techniques beyond those used in conventional processors and provides aggressive instruction prefetch and advanced branch prediction through a hierarchy of

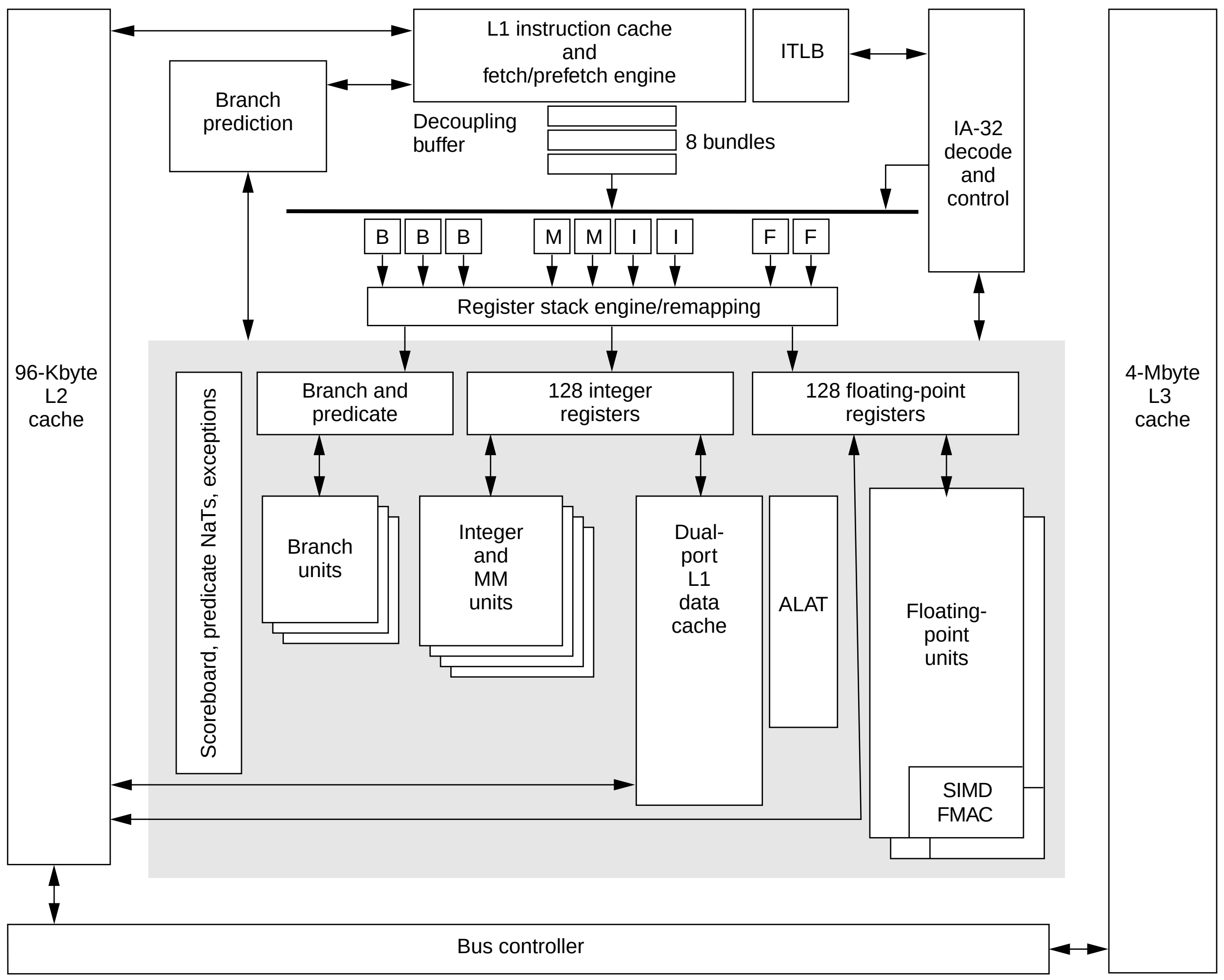


Figure 4. Itanium processor block diagram.

branch prediction structures. A decoupling buffer allows the front end to speculatively fetch ahead, further hiding instruction cache latency and branch prediction latency.

Block diagram

Figure 4 provides the block diagram of the Itanium processor. Figure 5 provides a die plot of the silicon database. A few top-level metal layers have been stripped off to create a suitable view.

Details of the core pipeline

The following describes details of the core processor microarchitecture.

Decoupled software-directed front end

Given the high execution rate of the processor (six instructions per clock), an aggressive front end is needed to keep the machine effectively fed, especially in the presence of disruptions due to branches and cache misses. The machine's front end is decoupled from the back end.

Acting in conjunction with sophisticated branch prediction and correction hardware, the machine speculatively fetches instructions from a moderate-size, pipelined instruction cache into a decoupling buffer. A hierarchy of branch predictors, aided by branch hints, provides up to four progressively improving instruction pointer resteers. Software-initiated prefetch probes for future misses in the instruction cache and then prefetches such target code from the level-2 (L2) cache into a streaming buffer and eventually into the

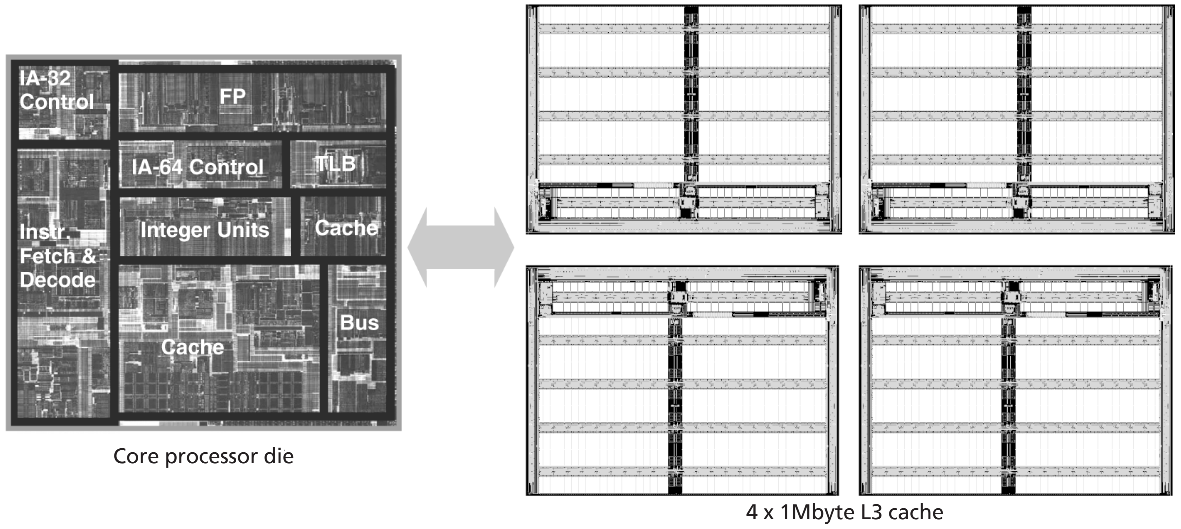


Figure 5. Die plot of the silicon database.

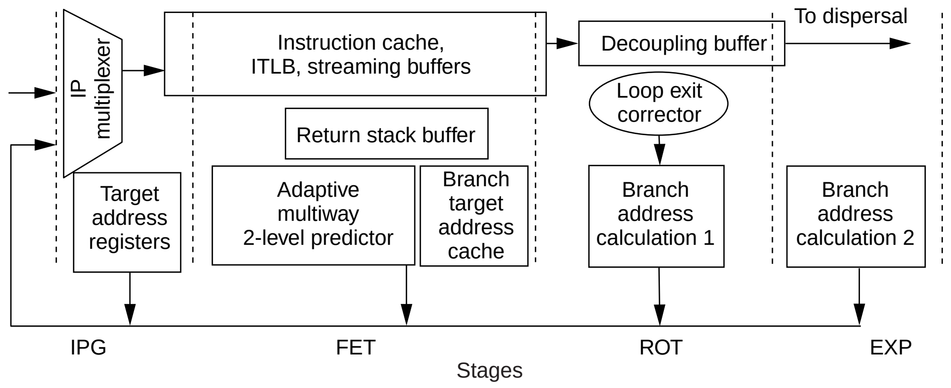

IP multiplexer
Instruction cache, ITLB, streaming buffers
Decoupling buffer
To dispersal
Loop exit corrector
Return stack buffer
Target address registers
Adaptive multiway 2-level predictor
Branch target address cache
Branch address calculation 1
Branch address calculation 2
IPG
FET
ROT
EXP
Stages

Figure 6. Processor front end.

instruction cache. Figure 6 illustrates the front-end microarchitecture.

Speculative fetches. The 16-Kbyte, four-way set-associative instruction cache is fully pipelined and can deliver 32 bytes of code (two instruction bundles or six instructions) every clock. The cache is supported by a single-cycle, 64-entry instruction translation look-aside buffer (TLB) that is fully associative and backed up by an on-chip hardware page walker.

The fetched code is fed into a decoupling buffer that can hold eight bundles of code. As a result of this buffer, the machine's front end can continue to fetch instructions into the buffer even when the back end stalls. Conversely, the buffer can continue to feed the back end even when the front end is disrupted by fetch bubbles due to branches or instruction cache misses.

Hierarchy of branch predictors. The processor employs a hierarchy of branch prediction structures to deliver high-accuracy and low-penalty predictions across a wide spectrum of workloads. Note that if a branch misprediction led to a full pipeline flush, there would be nine cycles of pipeline bubbles before the pipeline is full again. This would mean a heavy performance loss. Hence, we've placed significant emphasis on boosting the overall branch prediction rate as well as reducing the branch prediction and correction latency.

The branch prediction hardware is assisted by branch hint directives provided by the compiler (in the form of explicit branch predict, or BRP, instructions as well as hint specifiers on branch instructions). The directives provide branch target addresses, static hints on branch direction, as well as indications on when to use dynamic prediction. These directives are programmed into the branch prediction structures and used in conjunction with dynamic prediction schemes. The machine provides up to four progressive predictions and corrections to the fetch pointer, greatly reducing the likelihood of a full-pipeline flush due to a mispredicted branch.

- *Resteer 1: Single-cycle predictor.* A special set of four branch prediction registers (called target address registers, or TARs) provides single-cycle turnaround on certain branches (for example, loop branches in numeric code), operating under tight compiler control. The compiler programs these registers using BRP hints, distinguishing these hints with a special "importance" bit designator and indicating that these directives must get allocated into this small structure. When the instruction pointer of the candidate branch hits in these registers, the branch is predicted taken, and these registers provide the target address for the resteer. On such taken branches no bubbles appear in the execution schedule due to branching.
- *Resteer 2: Adaptive multiway and return predictors.* For scalar codes, the processor employs a dynamic, adaptive, two-level prediction scheme[3,4] to achieve well over 90% prediction rates on branch direction. The branch prediction table (BPT) contains 512 entries (128 sets $\times$ 4 ways). Each entry, selected by the branch address, tracks the four most recent occurrences of that branch. This 4-bit value then indexes into one of 128 pattern tables (one per set). The 16 entries in each pattern table use a 2-bit, saturating, up-down counter to predict branch direction.

  The branch prediction table structure is additionally enhanced for multiway branches with a 64-entry, multiway branch prediction table (MBPT) that employs a similar algorithm but keeps three history registers per bundle entry. A find-first-taken selection provides the first taken branch indication for the multiway branch bundle. Multiway branches are expected to be common in EPIC code, where multiple basic blocks are expected to collapse after use of speculation and predication.

  Target addresses for this branch resteer are provided by a 64-entry target address cache (TAC). This structure is updated by branch hints (using BRP and move-BR hint instructions) and also managed dynamically. Having the compiler program the structure with the upcoming footprint of the program is an advantage and enables a small, 64-entry structure to be effective even on large commercial workloads, saving die area and implementation complexity. The BPT and MBPT cause a front-end resteer only if the target address for the resteer is present in the target address cache. In the case of misses in the BPT and MBPT, a hit in the target address cache also provides a branch direction prediction of taken.

  A return stack buffer (RSB) provides predictions for return instructions. This buffer contains eight entries and stores return addresses along with corresponding register stack frame information.
- *Resteers 3 and 4: Branch address calculation and correction.* Once branch instruction opcodes are available (ROT stage), it's possible to apply a correction to predictions made earlier. The BAC1 stage applies a correction for the exit condition on modulo-scheduled loops through a special "perfect-loop-exit-predictor" structure that keeps track of the loop count extracted during the loop initialization code. Thus, loop exits should never see a branch misprediction in the back end. Additionally, in case of misses in the earlier prediction structures, BAC1 extracts static prediction information and addresses from branch instructions in the rightmost slot of a bundle and uses these to provide a correction. Since most templates will place a branch in the rightmost slot, BAC1 should handle most branches. BAC2 applies a more general correction for branches located in any slot.

Software-initiated prefetch. Another key element of the front end is its software-initiated instruction prefetch. Prefetch is triggered by prefetch hints (encoded in the BRP instructions as well as in actual branch instructions) as they pass through the ROT stage. Instructions get prefetched from the L2 cache into an instruction-streaming buffer (ISB) containing eight 32-byte entries. Support exists to prefetch either a short burst of 64 bytes of code (typically, a basic block residing in up to four bundles) or a long sequential instruction stream. Short burst prefetch is initiated by a BRP instruction hoisted well above the actual branch. For longer code streams, the sequential streaming ("many") hint from the branch instruction triggers a continuous stream of additional prefetch requests until a taken branch is encountered. The instruction cache filters prefetch requests. The cache tags and the TLB have been enhanced with an additional port to check whether an address will lead to a miss. Such requests are sent to the L2 cache.

The compiler can improve overall fetch performance by aggressive issue and hoisting of BRP instructions, and by issuing sequential prefetch hints on the branch instruction when branching to long sequential codes. To fully hide the latency of returns from the L2 cache, BRP instructions that initiate prefetch should be hoisted 12 fetch cycles ahead of the branch. Hoisting by five cycles breaks even with no prefetch at all. Every hoisted cycle above five cycles has the potential of shaving one fetch bubble. Although this kind of hoisting of BRP instructions is a tall order, it does provide a mechanism for the compiler to eliminate instruction fetch bubbles.

Efficient instruction and operand delivery

After instructions are fetched in the front end, they move into the middle pipeline that disperses instructions, implements the architectural renaming of registers, and delivers operands to the wide parallel hardware. The hardware resources in the back end of the machine are organized around nine issue ports. The instruction and operand delivery hardware maps the six incoming instructions onto the nine issue ports and remaps the virtual register identifiers specified in the source code onto physical registers used to access the register file. It then provides the source data to the execution core. The dispersal and renaming hardware exploits high-level semantic information provided by the IA-64 software, efficiently enabling greater ILP and reduced instruction path length.

Explicit parallelism directives. The instruction dispersal mechanism disperses instructions presented by the decoupling buffer to the processor's issue ports. The processor has a total of nine issue ports capable of issuing up to two

Machine resources per port

Tables A-C describe the vocabulary of operations supported on the different issue ports in the Itanium processor. The issue ports feed into memory (M), integer (I), floating-point (F), and branch (B) execution data paths.

Table A. Memory and integer execution resources.

	Ports for issuing an instruction				Latency
Instruction class	M0	M1	I0	I1	(no. of clock cycles)
ALU (add, shift-add, logical, addp4, compare)	•	•	•	•	
Sign/zero extend, move long			•	•	1
Fixed extract/deposit, Tbit, TNaT			•		1
Multimedia ALU (add/avg./etc.)	•	•	•	•	2
MM shift, avg, mix, pack			•	•	2
Move to/from branch/predicates/ ARs, packed multiply, pop count			•		2
Load/store/prefetch/setf/break.m/ cache control/memory fence	•	•			2+
Memory management/system/getf	•				2+

Table B. Floating-point execution resources.

	Ports for issuing an instruction		Latency
Instruction class	F0	F1	(no. of clock cycles)
FMAC, SIMD FMAC	•	•	5
Fixed multiply	•	•	7
FClrf	•	•	1
Fchk	•	•	1
Fcompare	•		2
Floating-point logicals, class, min/max, pack, select	•		5

Table C. Branch execution resources.

	Ports for issuing an instruction		
Instruction class	B0	B1	B2
Conditional or unconditional branch	•	•	•
Call/return/indirect	•	•	•
Loop-type branch, BSW, cover			•
RFI			•
BRP (branch hint)	•	•	•

Table 1. Instruction bundles capable of full-bandwidth dispersal.

First bundle*	Second bundle
MIH	MLI, MFI, MIB, MBB, or MFB
MFI or MLI	MLI, MFI, MIB, MBB, BBB, or MFB
MII	MBB, BBB, or MFB
MMI	BBB
MFH	MII, MLI, MFI, MIB, MBB, MFB

* B slots support branches and branch hints.

* H designates a branch hint operation in the B slot.

memory instructions (ports M0 and M1), two integer (ports I0 and I1), two floating-point (ports F0 and F1), and three branch instructions (ports B0, B1, and B2) per clock. The processor's 17 execution units are fed through the M, I, F, and B groups of issue ports.

The decoupling buffer feeds the dispersal in a bundle granular fashion (up to two bundles or six instructions per cycle), with a fresh bundle being presented each time one is consumed. Dispersal from the two bundles is instruction granular—the processor disperses as many instructions as can be issued (up to six) in left-to-right order. The dispersal algorithm is fast and simple, with instructions being dispersed to the first available issue port, subject to two constraints: detection of instruction independence and detection of resource oversubscription.

- *Independence.* The processor must ensure that all instructions issued in parallel are either independent or contain only allowed dependencies (such as a compare instruction feeding a dependent conditional branch). This question is easily dealt with by using the stop-bits feature of the IA-64 ISA to explicitly communicate parallel instruction semantics. Instructions between consecutive stop bits are deemed independent, so the instruction independence detection hardware is trivial. This contrasts with traditional RISC processors that are required to perform $O(n^2)$ (typically dozens) comparisons between source and destination register specifiers to determine independence.
- *Oversubscription.* The processor must also guarantee that there are sufficient execution resources to process all the instructions that will be issued in parallel. This oversubscription problem is facilitated by the IA-64 ISA feature of instruction bundle templates. Each instruction bundle not only specifies three instructions but also contains a 4-bit template field, indicating the type of each instruction: memory (M), integer (I), branch (B), and so on. By examining template fields from the two bundles (a total of only 8 bits), the dispersal logic can quickly determine the number of memory, integer, floating-point, and branch instructions incoming every clock. This is a hardware simplification resulting from the IA-64 instruction set architecture. Unlike conventional instruction set architectures, the instruction encoding itself doesn't need to be examined to determine the type of each operation. This feature removes decoders that would otherwise be required to examine many bits of the encoded instruction to determine the instruction's type and associated issue port.

  A second key advantage of the template-based dispersal strategy is that certain instruction types can only occur on specific locations within any bundle. As a result, the dispersal interconnection network can be significantly optimized; the routing required from dispersal to issue ports is roughly only half of that required for a fully connected crossbar.

Table 1 illustrates the effectiveness of the dispersal strategy by enumerating the instruction bundles that may be issued at full bandwidth. As can be seen, a rich mix of instructions can be issued to the machine at high throughput (six per clock). The combination of stop bits and bundle templates, as specified in the IA-64 instruction set, allows the compiler to indicate the independence and instruction-type information directly and effectively to the dispersal hardware. As a result, the hardware is greatly simplified, thereby allowing an efficient implementation of instruction dispersal to a wide execution core.

Efficient register remapping. After dispersal, the next step in preparing incoming instructions for execution involves implementing the register stacking and rotation functions.

Register stacking is an IA-64 technique that significantly reduces function call and return overhead. It ensures that all procedural input and output parameters are in specific register locations, without requiring the compiler to perform register-register or memory-register moves. On procedure calls, a fresh register frame is simply stacked on top of existing frames in the large register file, without the need for an explicit save of the caller's registers. This enables low-overhead procedure calls, providing significant performance benefit on codes that are heavy in calls and returns, such as those in object-oriented languages.

Register rotation is an IA-64 technique that allows very low overhead, software-pipelined loops. It broadens the applicability of compiler-driven software pipelining to a wide variety of integer codes. Rotation provides a form of register renaming that allows every iteration of a software-pipelined loop to have a fresh copy of loop variables. This is accomplished by accessing the registers through an indirection based on the iteration count.

Both stacking and rotation require the hardware to remap the register names. This remapping translates the incoming virtual register specifiers onto outgoing physical register specifiers, which are then used to perform the actual lookup of the various register files. Stacking can be thought of as simply adding an offset to the virtual register specifier. In a similar fashion, rotation can also be viewed as an offset-modulo add. The remapping function supports both stacking and rotation for the integer register specifiers, but only register rotation for the floating-point and predicate register specifiers.

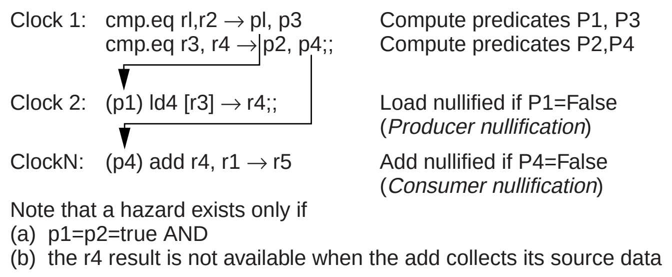
The Itanium processor efficiently supports the register remapping for both register stacking and rotation with a set of adders and multiplexers contained in the pipeline's REN stage. The stacking logic requires only one 7-bit adder for each specifier, and the rotation logic requires either one (predicate or floating-point) or two (integer) additional 7-bit adders. The extra adder on the integer side is needed due to the interaction of stacking with rotation. Therefore, for full six-syllable execution, a total of ninety-eight 7-bit adders and 42 multiplexers implement the combination of integer, floating-point, and predicate remapping for all incoming source and destination registers. The total area taken by this function is less than 0.25 square mm.

The register-stacking model also requires special handling when software allocates more virtual registers than are currently physically available in the register file. A special state machine, the register stack engine (RSE), handles this case—termed stack overflow. This engine observes all stacked register allocation or deallocation requests. When an overflow is detected on a procedure call, the engine silently takes control of the pipeline, spilling registers to a backing store in memory until sufficient physical registers are available. In a similar manner, the engine handles the converse situation—termed stack underflow—when registers need to be restored from a backing store in memory. While these registers are being spilled or filled, the engine simply stalls instructions waiting on the registers; no pipeline flushes are needed to implement the register spill/restore operations.

Register stacking and rotation combine to provide significant performance benefits for a variety of applications, at the modest cost of a number of small adders, an additional pipeline stage, and control logic for a programmer-invisible register stack engine.

Large, multiported register files. The processor provides an abundance of registers and execution resources. The 128-entry integer register file supports eight read ports and six write ports. Note that four ALU operations require eight read ports and four write ports from the register file, while pending load data returns need two additional write ports (two returns per cycle). The read and write ports can adequately support two memory and two integer instructions every clock. The IA-64 instruction set includes a feature known as postincrement. Here, the address register of a memory operation can be incremented as a side effect of the operation. This is supported by simply using two of the four ALU write ports. (These two ALUs and write ports would otherwise have been idle when memory operations are issued off their ports).

The floating-point register file also consists of 128 registers, supports double extended-precision arithmetic, and can sustain two memory ports in parallel with two multiply-accumulate units. This combination of



```
Clock 1:  cmp.eq rl,r2 → pl, p3          Compute predicates P1, P3
          cmp.eq r3, r4 → p2, p4;;        Compute predicates P2,P4

Clock 2:  (p1) ld4 [r3] → r4;;           Load nullified if P1=False
                                          (Producer nullification)

ClockN:   (p4) add r4, r1 → r5           Add nullified if P4=False
                                          (Consumer nullification)
Note that a hazard exists only if
(a)  p1=p2=true AND
(b)  the r4 result is not available when the add collects its source data
```

Figure 7. Predicated producer-consumer dependencies.

resources requires eight read and four write ports. The register write ports are separated in even and odd banks, allowing each memory return to update a pair of floating-point registers.

The other large register file is the predicate register file. This register file has several unique characteristics: each entry is 1 bit, it has many read and write ports (15 reads/11 writes), and it supports a "broadside" read or write of the entire register file. As a result, it has a distinct implementation, as described in the "Implementing predication elegantly" section (next page).

High ILP execution core

The execution core is the heart of the EPIC implementation. It supports data-speculative and control-speculative execution, as well as predicated execution and the traditional functions of hazard detection and branch execution. Furthermore, the processor's execution core provides these capabilities in the context of the wide execution width and powerful instruction semantics that characterize the EPIC design philosophy.

Stall-based scoreboard control strategy. As mentioned earlier, the frequency target of the Itanium processor was governed by several key timing paths such as the ALU plus bypass and the two-cycle data cache. All of the control paths within the core pipeline fit within the given cycle time—detecting and dealing with data hazards was one such key control path.

To achieve high performance, we adopted a nonblocking cache with a scoreboard-based stall-on-use strategy. This is particularly valuable in the context of speculation, in which certain load operations may be aggressively boosted to avoid cache miss latencies, and the resulting data may potentially not be consumed. For such cases, it is key that 1) the pipeline not be interrupted because of a cache miss, and 2) the pipeline only be interrupted if and when the unavailable data is needed.

Thus, to achieve high performance, the strategy for dealing with detected data hazards is based on stalls—the pipeline only stalls when unavailable data is needed and stalls only as long as the data is unavailable. This strategy allows the entire processor pipeline to remain filled, and the in-flight dependent instructions to be immediately ready to continue as soon as the required data is available. This contrasts with other high-frequency designs, which are based on flushing and require that the pipeline be emptied when a hazard is detected, resulting in reduced performance. On the Itanium processor, innovative techniques reap the performance benefits of a stall-based strategy and yet enable high-frequency operation on this wide machine.

The scoreboard control is also enhanced to support predication. Since most operations within the IA-64 instruction set architecture can be predicated, either the producer or the consumer of a given piece of data may be nullified by having a false predicate. Figure 7 illustrates an example of such a case. Note that if either the producer or consumer operation is nullified via predication, there are no hazards. The processor scoreboard therefore considers both the producer and consumer predicates, in addition to the normal operand availability, when evaluating whether a hazard exists. This hazard evaluation occurs in the REG (register read) pipeline stage.

Given the high frequency of the processor pipeline, there's not sufficient time to both compute the existence of a hazard, and effect a global pipeline stall in a single clock cycle. Hence, we use a unique deferred-stall strategy. This approach allows any dependent consumer instructions to proceed from the REG into the EXE (execute) pipeline stage, where they are then stalled—hence the term deferred stall.

However, the instructions in the EXE stage no longer have read port access to the register file to obtain new operand data. Therefore, to ensure that the instructions in the EXE stage procure the correct data, the latches at the start of the EXE stage (which contain the source data values) continuously snoop all returning data values, intercepting any data that the instruction requires. The logic used to perform this data interception is identical to the register bypass network used to collect operands for instructions in the REG stage. By noting that instructions observing a deferred stall in the REG stage don't require the use of the bypass network, the EXE stage instructions can usurp the bypass network for the deferred stall. By reusing existing register bypass hardware, the deferred stall strategy is implemented in an area-efficient manner. This allows the processor to combine the benefits of high frequency with stall-based pipeline control, thereby precluding the penalty of pipeline flushes due to replays on register hazards.

Execution resources. The processor provides an abundance of execution resources to exploit ILP. The integer execution core includes two memory and two integer ports, with all four ports capable of executing arithmetic, shift-and-add, logical, compare, and most integer SIMD multimedia operations. The memory ports can also perform load and store operations, including loads and stores with postincrement functionality. The integer ports add the ability to perform the less-common integer instructions, such as test bit, look for zero byte, and variable shift. Additional uncommon instructions are also implemented on only the first integer port.

See the earlier sidebar for a full enumeration of the per-port capabilities and associated instruction latencies on the processor. In general, we designed the method used to map instructions onto each port to maximize overall performance, by balancing the instruction frequency with the area and timing impact of additional execution resources.

Implementing predication elegantly. Predication is another key feature of the IA-64 architecture, allowing higher performance by eliminating branches and their associated misprediction penalties.[5] However, predication affects several key aspects of the pipeline design. Predication turns a control dependency (branching on the condition) into a data dependency (execution and forwarding of data dependent upon the value of the predicate). If spurious stalls and pipeline disruptions get introduced during predicated execution, the benefit of branch misprediction elimination will be squandered. Care was taken to ensure that predicates are implemented transparently in the pipeline.

The basic strategy for predicated execution is to allow all instructions to read the register file and get issued to the hardware regardless of their predicate value. Predicates are used to configure the data-forwarding network, detect the presence of hazards, control pipeline advances, and conditionally nullify the execution and retirement of issued operations. Predicates also feed the branching hardware. The predicate register file is a highly multiported structure. It is accessed in parallel with the general registers in the REG stage. Since predicates themselves are generated in the execution core (from compare instructions, for

IA-32 compatibility

Another key feature of the Itanium processor is its full support of the IA-32 instruction set in hardware (see Figure A). This includes support for running a mix of IA-32 applications and IA-64 applications on an IA-64 operating system, as well as IA-32 applications on an IA-32 operating system, in both uniprocessor and multiprocessor configurations. The IA-32 engine makes use of the EPIC machine's registers, caches, and execution resources. To deliver high performance on legacy binaries, the IA-32 engine dynamically schedules instructions.[1,2] The IA-64 Seamless Architecture is defined to enable running IA-32 system functions in native IA-64 mode, thus delivering native performance levels on the system functionality.

References

1. R. Colwell and R. Steck, "A 0.6μm BICMOS Microprocessor with Dynamic Execution," *Proc. Int'l Solid-State Circuits Conf.,* IEEE Press, Piscataway, N.J., 1995, pp. 176-177.
2. D. Papworth, "Tuning the Pentium Pro Microarchitecture," *IEEE Micro,* Mar./Apr. 1996, pp. 8-15.

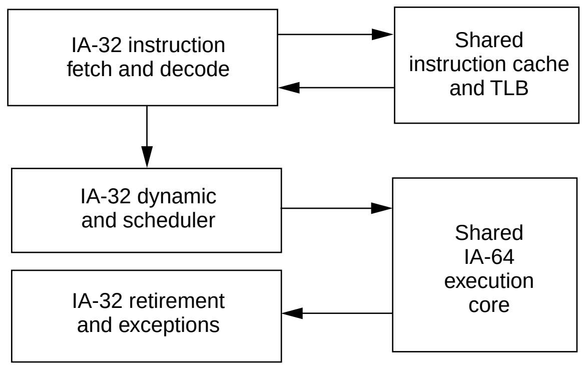

IA-32 instruction fetch and decode
Shared instruction cache and TLB
IA-32 dynamic and scheduler
Shared IA-64 execution core
IA-32 retirement and exceptions

Figure A. IA-32 compatibility microarchitecture.

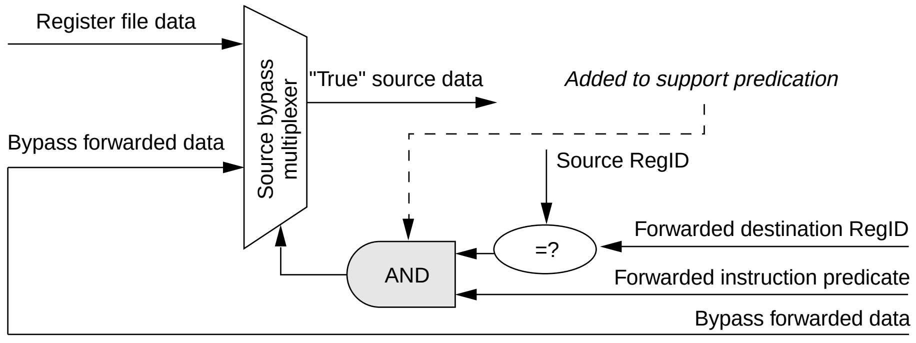

Register file data
Source bypass multiplexer
"True" source data
Added to support predication
Bypass forwarded data
Source RegID
=?
Forwarded destination RegID
AND
Forwarded instruction predicate
Bypass forwarded data

Figure 8. Predicated bypass control.

example) and may be in flight when they're needed, they must be forwarded quickly to the specific hardware that consumes them.

Note that predication affects the hazard detection logic by nullifying either data producer or consumer instructions. Consumer nullification is performed after reading the predicate register file (PRF) for the predicate sources of the six instructions in the REG pipeline stage. Producer nullification is performed after reading the predicate register file for the predicate sources for the six instructions in the EXE stage.

Finally, three conditional branches can be executed in the DET pipeline stage; this requires reading three additional predicate sources. Thus, a total of 15 read ports are needed to access the predicate register file. From a write port perspective, 11 predicates can be written every clock: eight from four parallel integer compares, two from a floating-point compare, and one via the stage predicate write feature of loop branches. These read and write ports are in addition to a broadside read and write capability that allows a single instruction to read or write the entire 64-entry predicate register into or from a single 64-bit integer register. The predicate register file is implemented as a single 64-bit latch with 15 simple 64:1 multiplexers being used as the read ports. Similarly, the 11 write ports are efficiently implemented, with each being a 6:64 decoder, with an AND-OR structure used to update the actual predicate register file latch. Broadside reads and writes are easily implemented by reading or writing the contents of the entire 64 bit latch.

In-flight predicates must be forwarded quickly after generation to the point of consumption. The costly bypass logic that would have been needed for this is eliminated by taking advantage of the fact that all predicate-writing instructions have deterministic latency. Instead, a speculative predicate register file (SPRF) is used and updated as soon as predicate data is computed. The source predicate of any dependent instruction is then read directly from this register file, obviating the need for bypass logic. A separate architectural predicate register file (APRF) is only updated when a predicate-writing instruction retires and is only then allowed to update the architectural state.

In case of an exception or pipeline flush, the SPRF is copied from the APRF in the shadow of the flush latency, undoing the effect of any misspeculative predicate writes. The combination of latch-based implementation and the two-file strategy allow an area-efficient and timing-efficient implementation of the highly ported predicate registers.

Figure 8 shows one of the six EXE stage predicates that allow or nullify data forwarding in the data-forwarding network. The other five predicates are handled identically. Predication control of the bypass network is implemented very efficiently by ANDing the predicate value with the destination-valid signal present in conventional bypass logic networks. Instructions with false predicates are treated as merely not writing to their destination register. Thus, the impact of predication on the operand-forwarding network is fairly minimal.

Optimized speculation support in hardware. With minimal hardware impact, the Itanium processor enables software to hide the latency of load instructions and their dependent uses by boosting them out of their home basic block. This is termed speculation. To perform effective speculation, two key issues must be addressed. First, any exceptions that are detected must be deferrable until an operation's home basic block is encountered; this is termed control speculation. Second, all stores between the boosted load and its home location must be checked for address overlap. If there is an overlap, the latest store should forward the correct data; this is termed data speculation. The Itanium processor provides effective support for both forms of speculation.

In case of control speculation, normal exception checks are performed for a control-speculative load instruction. In the common case, no exception is encountered, and therefore no special handling is required. On a detected exception, the hardware examines the exception type, software-managed architectural control registers, and page attributes to determine whether the exception should be handled immediately (such as for a TLB miss) or deferred for future handling.

For a deferral, a special deferred exception token called NaT (Not a Thing) bit is retained for each integer register, and a special floating-point value, called NaTVal and encoded in the NaN space, is set for floating-point registers. This token indicates that a deferred exception was detected. The deferred exception token is then propagated into result registers when any of the source registers indicates such a token. The exception is reported when either a speculation check or nonspeculative use (such as a store instruction) consumes a register that is flagged with the deferred exception token. In this way, NaT generation leverages traditional exception logic simply, and NaT propagation uses straightforward data path logic.

The existence of NaT bits and NaTVals also affect the register spill-and-fill logic. For explicit software-driven register spills and fills, special move instructions (store.spill and load.fill) are supported that don't take exceptions when encountering NaT'ed data. For floating-point data, the entire data is simply moved to and from memory. For integer data, the extra NaT bit is written into a special register (called UNaT, or user NaT) on spills, and is read back on the load.fill instruction. The UNaT register can also be written to memory if more than 64 registers need to be spilled. In the case of implicit spills and fills generated by the register save engine, the engine collects the NaT bits into another special register (called RNaT, or register NaT), which is then spilled (or filled) once for every 64 register save engine stores (or loads).

For data speculation, the software issues an advanced load instruction. When the hardware encounters an advanced load, it places the address, size, and destination register of the load into the ALAT structure. The ALAT then observes all subsequent explicit store instructions, checking for overlaps of the valid advanced load addresses present in the ALAT. In the common case, there's no match, the ALAT state is unchanged, and the advanced load result is used normally. In the case of an overlap, all address-matching advanced loads in the ALAT are invalidated.

With minimal hardware impact, the Itanium processor enables software to hide the latency of load instructions and their dependent uses by boosting them out of their home basic block.

After the last undisambiguated store prior to the load's home basic block, an instruction can query the ALAT and find that the advanced load was matched by an intervening store address. In this situation recovery is needed. When only the load and no dependent instructions were boosted, a load-check (ld.c) instruction is used, and the load instruction is reissued down the pipeline, this time retrieving the updated memory data. As an important performance feature, the ld.c instruction can be issued in parallel with instructions dependent on the load result data. By allowing this optimization, the critical load uses can be issued immediately, allowing the ld.c to effectively be a zero-cycle operation. When the advanced load and its dependent uses were boosted, an advanced check-load (chk.a) instruction traps to a user-specified handler for a special fix-up code that reissues the load instruction and the operations dependent on the load. Thus, support for data speculation was added to the pipeline

Floating-point feature set

The FPU in the processor is quite advanced. The native 82-bit hardware provides efficient support for multiple numeric programming models, including support for single, double, extended, and mixed-mode-precision computations. The wide-range 17-bit exponent enables efficient support for extended-precision library functions as well as fast emulation of quad-precision computations. The large 128-entry register file provides adequate register resources. The FPU execution hardware is based on the floating-point multiply-add (FMAC) primitive, which is an effective building block for scientific computation.[1] The machine provides execution hardware for four double-precision or eight single-precision flops per clock. This abundant computation bandwidth is balanced with adequate operand bandwidth from the registers and memory subsystem. With judicious use of data prefetch instructions, as well as cache locality and allocation management hints, the software can effectively arrange the computation for sustained high utilization of the parallel hardware.

FMAC units

The FPU supports two fully pipelined, 82-bit FMAC units that can execute single, double, or extended-precision floating-point operations. This delivers a peak of 4 double-precision flops/clock, or 3.2 Gflops at 800 MHz. FMAC units execute FMA, FMS, FNMA, FCVTFX, and FCVTXF operations. When bypassed to one another, the latency of the FMAC arithmetic operations is five clock cycles.

The processor also provides support for executing two SIMD-floating-point instructions in parallel. Since each instruction issues two single-precision FMAC operations (or four single-precision flops), the peak execution bandwidth is 8 single-precision flops/clock or 6.4 Gflops at 800 MHz. Two supplemental single-precision FMAC units support this computation. (Since the read of an 82-bit register actually yields two single-precision SIMD operands, the second operand in each case is peeled off and sent to the supplemental SIMD units for execution.) The high computational rate on single precision is especially suitable for digital content creation workloads.

The divide operation is done in software and can take advantage of the twin fully pipelined FMAC hardware. Software-pipelined divide operations can yield high throughput on division and square-root operations common in 3D geometry codes.

The machine also provides one hardware pipe for execution of FCMPs and other operations (such as FMERGE, FPACK, FSWAP, FLogicals, reciprocal, and reciprocal square root). Latency of the FCMP operations is two clock cycles; latency of the other floating-point operations is five clock cycles.

Operand bandwidth

Care has been taken to ensure that the high computational bandwidth is matched with operand feed bandwidth. See Figure B. The 128-entry floating-point register file has eight read and four write ports. Every cycle, the eight read ports can feed two extended-precision FMACs (each with three operands) as well as two floating-point stores to memory. The four write ports can accommodate two extended-precision results from the two FMAC units and the results from two load instructions each clock. To increase the effective write bandwidth into the FPU from memory, we divided the floating-point registers into odd and even banks. This enables the two physical write ports dedicated to load returns to be used to write four values per clock to the register file (two to each bank), using two ldf-pair instructions. The ldf-pair instructions must obey the restriction that the pair of consecutive memory operands being loaded in sends one operand to an even register and the other to an odd register for proper use of the banks.

The earliest cache level to feed the FPU is the unified L2 cache (96 Kbytes). Two ldf-pair instructions can load four double-precision values from the L2 cache into the registers. The latency of loads from this cache to the FPU is nine clock cycles. For data beyond the L2 cache, the bandwidth to the L3 cache is two double-precision operations/clock (one 64-byte line every four clock cycles).

Obviously, to achieve the peak rating of four double-precision floating-point operations per clock cycle, one needs to feed the FMACs with six operands per clock. The L2 memory can feed a peak of four operands per clock. The remaining two need to come from the register file. Hence, with the right amount of data reuse, and with appropriate cache management strategies aimed at ensuring that the L2 cache is well primed to feed the FPU, many workloads can deliver sustained performance at near the peak floating-point operation rating. For data without locality, use of the NT2 and NTA hints enables the data to appear to virtually stream into the FPU through the next level of memory.

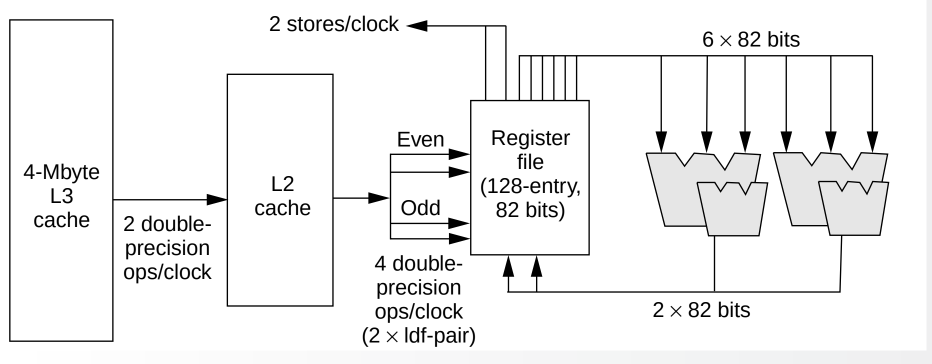

4-Mbyte L3 cache
2 double-precision ops/clock
L2 cache
Even
Odd
4 double-precision ops/clock (2 × ldf-pair)
Register file (128-entry, 82 bits)
2 stores/clock
6 × 82 bits
2 × 82 bits

Figure B. FMAC units deliver 8 flops/clock.

FPU and integer core coupling

The floating-point pipeline is coupled to the integer pipeline. Register file read occurs in the REG stage, with seven stages of execution extending beyond the REG stage, followed by floating-point write back. Safe instruction recognition (SIR) hardware enables delivery of precise exceptions on numeric computation. In the FP1 (or EXE) stage, an early examination of operands is performed to determine the possibility of numeric exceptions on the instructions being issued. If the instructions are unsafe (have potential for raising exceptions), a special form of hardware microreplay is incurred. This mechanism enables instructions in the floating-point and integer pipelines to flow freely in all situations in which no exceptions are possible.

The FPU is coupled to the integer data path via transfer paths between the integer and floating-point register files. These transfers (setf, getf) are issued on the memory ports and made to look like memory operations (since they need register ports on both the integer and floating-point registers). While setf can be issued on either M0 or M1 ports, getf can only be issued on the M0 port. Transfer latency from the FPU to the integer registers (getf) is two clocks. The latency for the reverse transfer (setf) is nine clocks, since this operation appears like a load from the L2 cache.

We enhanced the FPU to support integer multiply inside the FMAC hardware. Under software control, operands are transferred from the integer registers to the FPU using setf. After multiplication is complete, the result is transferred to the integer registers using getf. This sequence takes a total of 18 clocks (nine for setf, seven for fmul to write the registers, and two for getf). The FPU can execute two integer multiply-add (XMA) operations in parallel. This is very useful in cryptographic applications. The presence of twin XMA pipelines at 800 MHz allows for over 1,000 decryptions per second on a 1,024-bit RSA using private keys (server-side encryption/decryption).

FPU controls

The FPU controls for operating precision and rounding are derived from the floating-point status register (FPSR). This register also contains the numeric execution status of each operation. The FPSR also supports speculation in floating-point computation. Specifically, the register contains four parallel fields or tracks for both controls and flags to support three parallel speculative streams, in addition to the primary stream.

Special attention has been placed on delivering high performance for speculative streams. The FPU provides high throughput in cases where the FCLRF instruction is used to clear status from speculative tracks before forking off a fresh speculative chain. No stalls are incurred on such changes. In addition, the FCHKF instruction (which checks for exceptions on speculative chains on a given track) is also supported efficiently. Interlocks on this instruction are track-granular, so that no interlock stalls are incurred if floating-point instructions in the pipeline are only targeting the other tracks. However, changes to the control bits in the FPSR (made via the FSETC instruction or the MOV GR→FPSR instruction) have a latency of seven clock cycles.

FPU summary

The FPU feature set is balanced to deliver high performance across a broad range of computational workloads. This is achieved through the combination of abundant execution resources, ample operand bandwidth, and a rich programming environment.

References

1. B.Olsson et al., “RISC System/6000 Floating-Point Unit,” *IBM RISC System/6000 Technology,* IBM Corp., 1990, pp. 34-42.

in a straightforward manner, only needing management of a small ALAT in hardware.

As shown in Figure 9, the ALAT is implemented as a 32-entry, two-way set-associative structure. The array is looked up based on the advanced load’s destination register ID, and each entry contains an advanced load’s physical address, a special octet mask, and a valid bit. The physical address is used to compare against subsequent stores, with the octet mask bits used to track which bytes have actually been advance loaded. These are used in case of partial overlap or in cases where the load and store are different sizes. In case of a match, the corresponding valid bit is cleared. The later check instruction then simply queries the ALAT to examine if a valid ALAT entry still exists for the ALAT.

The combination of a simple ALAT for the data speculation, in conjunction with NaT bits and small changes to the exception logic for control speculation, eliminates the two fundamental barriers that software has traditionally encountered when boosting instructions. By adding this modest hardware

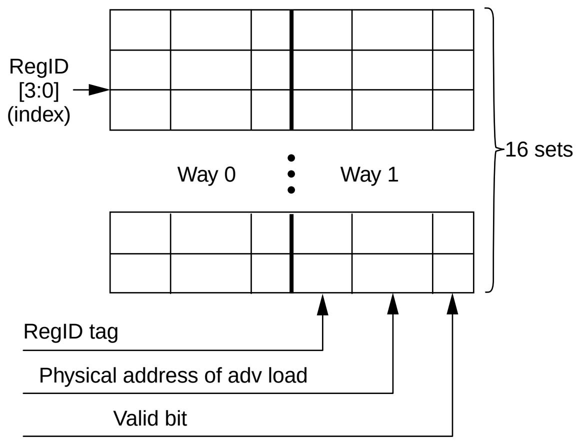


Figure 9. ALAT organization.

support for speculation, the processor allows the software to take advantage of the compiler's large scheduling window to hide memory latency, without the need for complex dynamic scheduling hardware.

Parallel zero-latency delay-executed branching. Achieving the highest levels of performance requires a robust control flow mechanism. The processor's branch-handling strategy is based on three key directions. First, branch semantics providing more program details are needed to allow the software to convey complex control flow information to the hardware. Second, aggressive use of speculation and predication will progressively lead to an emptying out of basic blocks, leaving clusters of branches. Finally, since the data flow from compare to dependent branch is often very tight, special care needs to be taken to enable high performance for this important case. The processor optimizes across all three of these fronts.

The processor efficiently implements the powerful branch vocabulary of the IA-64 instruction set architecture. The hardware takes advantage of the new semantics for improved branch handling. For example, the loop count (LC) register indicates the number of iterations in a For-type loop, and the epilogue count (EC) register indicates the number of epilogue stages in a software-pipelined loop.

By using the loop count information, high performance can be achieved by software pipelining all loops. Moreover, the implementation avoids pipeline flushes for the first and last loop iterations, since the actual number of iterations is effectively communicated to the hardware. By examining the epilog count register information, the processor automatically generates correct stage predicates for the epilogue iterations of the software-pipelined loop. This step leverages the predicate-remapping hardware along with the branch prediction information from the loop count register-based branch predictor.

Unlike conventional processors, the Itanium processor can execute up to three parallel branches per clock. This is implemented by examining the three controlling conditions (either predicates or the loop count/epilog count counter values) for the three parallel branches, and performing a priority encode to determine the earliest taken branch. All side effects of later instructions are automatically squashed within the branch execution unit itself, preventing any architectural state update from branches in the shadow of a taken branch. Given that the powerful branch prediction in the front end contains tailored support for multiway branch prediction, minimal pipeline disruptions can be expected due to this parallel branch execution.

Finally, the processor optimizes for the common case of a very short distance between the branch and the instruction that generates the branch condition. The IA-64 instruction set architecture allows a conditional branch to be issued concurrently with the integer compare that generates its condition code—no stop bit is needed. To accommodate this important performance optimization, the processor pipelines the compare-branch sequence. The compare instruction is performed in the pipeline's EXE stage, with the results being known by the end of the EXE clock. To accommodate the delivery of this condition to the branch hardware, the processor executes all branches in the DET stage. (Note that the presence of the DET stage isn't an overhead needed solely from branching. This stage is also used for exception collection and prioritization, and for the second clock of execution for integer-SIMD operations.) Thus, any branch issued in parallel with the compare that generates the condition will be evaluated in the DET stage, using the predicate results created in the previous (EXE) stage. In this manner, the processor can easily handle the case of compare and dependent branches issued in parallel.

As a result of branch execution in the DET stage, in the rare case of a full pipeline flush due to a branch misprediction, the processor will incur a branch misprediction penalty of nine pipeline bubbles. Note that we expect this to occur rarely, given the aggressive multitier branch prediction strategy in the front end. Most branches should be predicted correctly using one of the four progressive resteers in the front end.

The combination of enhanced branch semantics, three-wide parallel branch execution, and zero-cycle compare-to-branch latency allows the processor to achieve high performance on control-flow-dominated codes, in addition to its high performance on

Table 2. Implementation of cache hints.

Hint	Semantics	L1 response	L2 response	L3 response
NTA	Nontemporal (all levels)	Don't allocate	Allocate, mark as next replace	Don't allocate
NT2	Nontemporal (2 levels)	Don't allocate	Allocate, mark as next replace	Normal allocation
NT1	Nontemporal (1 level)	Don't allocate	Normal allocation	Normal allocation
T1 (default)	Temporal	Normal allocation	Normal allocation	Normal allocation
Bias	Intent to modify	Normal allocation	Allocate into exclusive state	Allocate into exclusive state

more computation-oriented data-flow-dominated workloads.

Memory subsystem

In addition to the high-performance core, the Itanium processor provides a robust cache and memory subsystem, which accommodates a variety of workloads and exploits the memory hints of the IA-64 ISA.

Three levels of on-package cache

The processor provides three levels of on-package cache for scalable performance across a variety of workloads. At the first level, instruction and data caches are split, each 16 Kbytes in size, four-way set-associative, and with a 32-byte line size. The dual-ported data cache has a load latency of two cycles, is write-through, and is physically addressed and tagged. The L1 caches are effective on moderate-size workloads and act as a first-level filter for capturing the immediate locality of large workloads.

The second cache level is 96 Kbytes in size, is six-way set-associative, and uses a 64-byte line size. The cache can handle two requests per clock via banking. This cache is also the level at which ordering requirements and semaphore operations are implemented. The L2 cache uses a four-state MESI (modified, exclusive, shared, and invalid) protocol for multiprocessor coherence. The cache is unified, allowing it to service both instruction and data side requests from the L1 caches. This approach allows optimal cache use for both instruction-heavy (server) and data-heavy (numeric) workloads. Since floating-point workloads often have large data working sets and are used with compiler optimizations such as data blocking, the L2 cache is the first point of service for floating-point loads. Also, because floating-point performance requires high bandwidth to the register file, the L2 cache can provide four double-precision operands per clock to the floating-point register file, using two parallel floating-point load-pair instructions.

The third level of on-package cache is 4 Mbytes in size, uses a 64-byte line size, and is four-way set-associative. It communicates with the processor at core frequency (800 MHz) using a 128-bit bus. This cache serves the large workloads of server- and transaction-processing applications, and minimizes the cache traffic on the frontside system bus. The L3 cache also implements a MESI protocol for microprocessor coherence.

A two-level hierarchy of TLBs handles virtual address translations for data accesses. The hierarchy consists of a 32-entry first-level and 96-entry second-level TLB, backed by a hardware page walker.

Optimal cache management

To enable optimal use of the cache hierarchy, the IA-64 instruction set architecture defines a set of memory locality hints used for better managing the memory capacity at specific hierarchy levels. These hints indicate the temporal locality of each access at each level of hierarchy. The processor uses them to determine allocation and replacement strategies for each cache level. Additionally, the IA-64 architecture allows a bias hint, indicating that the software intends to modify the data of a given cache line. The bias hint brings a line into the cache with ownership, thereby optimizing the MESI protocol latency.

Table 2 lists the hint bits and their mapping to cache behavior. If data is hinted to be nontemporal for a particular cache level, that data is simply not allocated to the cache. (On the L2 cache, to simplify the control logic, the processor implements this algorithm approximately. The data can be allocated to the cache, but the least recently used, or LRU, bits are modified to mark the line as the next target for replacement.) Note that the nearest cache level to feed

The Itanium processor is the first IA-64 processor and is designed to meet the demanding needs of a broad range of enterprise and scientific workloads.

the floating-point unit is the L2 cache. Hence, for floating-point loads, the behavior is modified to reflect this shift (an NT1 hint on a floating-point access is treated like an NT2 hint on an integer access, and so on).

Allowing the software to explicitly provide high-level semantics of the data usage pattern enables more efficient use of the on-chip memory structures, ultimately leading to higher performance for any given cache size and access bandwidth.

System bus

The processor uses a multidrop, shared system bus to provide four-way glueless multiprocessor system support. No additional bridges are needed for building up to a four-way system. Systems with eight or more processors are designed through clusters of these nodes using high-speed interconnects. Note that multidrop buses are a cost-effective way to build high-performance four-way systems for commercial transaction processing and e-business workloads. These workloads often have highly shared writeable data and demand high throughput and low latency on transfers of modified data between caches of multiple processors.

In a four-processor system, the transaction-based bus protocol allows up to 56 pending bus transactions (including 32 read transactions) on the bus at any given time. An advanced MESI coherence protocol helps in reducing bus invalidation transactions and in providing faster access to writeable data. The cache-to-cache transfer latency is further improved by an enhanced "defer mechanism," which permits efficient out-of-order data transfers and out-of-order transaction completion on the bus. A deferred transaction on the bus can be completed without reusing the address bus. This reduces data return latency for deferred transactions and efficiently uses the address bus. This feature is critical for scalability beyond four-processor systems.

The 64-bit system bus uses a source-synchronous data transfer to achieve 266-Mtransfers/s, which enables a bandwidth of 2.1 Gbytes/s. The combination of these features makes the Itanium processor system a scalable building block for large multiprocessor systems.

The Itanium processor is the first IA-64 processor and is designed to meet the demanding needs of a broad range of enterprise and scientific workloads. Through its use of EPIC technology, the processor fundamentally shifts the balance of responsibilities between software and hardware. The software performs global scheduling across the entire compilation scope, exposing ILP to the hardware. The hardware provides abundant execution resources, manages the bookkeeping for EPIC constructs, and focuses on dynamic fetch and control flow optimizations to keep the compiled code flowing through the pipeline at high throughput. The tighter coupling and increased synergy between hardware and software enable higher performance with a simpler and more efficient design.

Additionally, the Itanium processor delivers significant value propositions beyond just performance. These include support for 64 bits of addressing, reliability for mission-critical applications, full IA-32 instruction set compatibility in hardware, and scalability across a range of operating systems and multiprocessor platforms. MICRO

References

1. L. Gwennap, "Merced Shows Innovative Design," *Microprocessor Report,* MicroDesign Resources, Sunnyvale, Calif., Oct. 6, 1999, pp. 1, 6-10.
2. M.S. Schlansker and B.R. Rau, "EPIC: Explicitly Parallel Instruction Computing," *Computer,* Feb. 2000, pp. 37-45.
3. T.Y. Yeh and Y.N. Patt, "Two-Level Adaptive Training Branch Prediction," *Proc. 24th Ann. Int'l Symp. Microarchitecture,* ACM Press, New York, Nov. 1991, pp. 51-61.

4. L. Gwennap, "New Algorithm Improves Branch Prediction," *Microprocessor Report,* Mar. 27, 1995, pp. 17-21.
5. B.R. Rau et al., "The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-Offs," *Computer,* Jan. 1989, pp. 12-35.

Harsh Sharangpani was Intel's principal microarchitect on the joint Intel-HP IA-64 EPIC ISA definition. He managed the microarchitecture definition and validation of the EPIC core of the Itanium processor. He has also worked on the 80386 and i486 processors, and was the numerics architect of the Pentium processor. Sharangpani received an MSEE from the University of Southern California, Los Angeles and a BSEE from the Indian Institute of Technology, Bombay. He holds 20 patents in the field of microprocessors.

Ken Arora was the microarchitect of the execution and pipeline control of the EPIC core of the Itanium processor. He participated in the joint Intel/HP IA-64 EPIC ISA definition and helped develop the initial IA-64 simulation environment. Earlier, he was a designer and architect on the i486 and Pentium processors. Arora received a BS degree in computer science and a master's degree in electrical engineering from Rice University. He holds 12 patents in the field of microprocessor architecture and design.

Direct questions about this article to Harsh Sharangpani, Intel Corporation, Mail Stop SC 12-402, 2200 Mission College Blvd., Santa Clara, CA 95052; harsh.sharangpani@intel.com.