Itanium™ Architecture Overview

Gautam Doshi
Architect
Enterprise Processor Division
Intel Corporation

U.C. Berkeley Workshop
Dec 11, 2000
Agenda

- Today’s Architecture Challenges
  - Novel Itanium Architecture Features
  - Itanium™ Processor Micro-Architecture
  - Examples of Itanium Features in Action
Architecture Challenges

- Sequentiality inherent in traditional architectures
- Complex hardware needed to (re)extract ILP
- Limited ILP available within basic blocks
- Branches make extracting ILP difficult
- Memory dependencies further limit ILP
- Increasing latency exacerbates ILP need
- Limited resources: A fundamental constraint
- Shared resources create more overhead
- Loop ILP extraction costs code size
- And the challenges continue ...

Itanium overcomes these fundamental challenges!
Agenda

- Today’s Architecture Challenges
- Novel Itanium Architecture Features
- Itanium™ Processor Micro-Arch. Overview
- Examples of Itanium Features in Action
Itanium Architecture Features

- It’s all about **Parallelism**!
  - Enabling it
  - Enhancing it
  - Expressing it
  - Exploiting it

... at the proc./thread level for programmer

... at the instruction level for compiler

Enable, Enhance, Express, Exploit - Parallelism
Itanium Architecture
Performance Features

- Explicitly Parallel Instruction Semantics
- Predication and Control/Data Speculation
- Massive, Massive Resources (regs, mem)
- Register Stack and its Engine (RSE)
- Memory hierarchy management support
- Software Pipelining Support
- ...

Challenges addressed from the ground up
Explicitly Parallel Semantics

- Program = Sequence of **Parallel** Inst. Groups
- Implied order of instruction groups
- **NO** dependence between insts. within group
- So ...
  - High performance needs parallel execution
  - Parallel execution needs **independent** insts.
  - Independent instructions explicitly indicated

Parallelism inherent in Itanium architecture
Explicitly Parallel Semantics ...

**Dependent**
- `add r1 = r2, r3 ;;`
- `sub r4 = r1, r2 ;;`
- `shl r5 = r4, r8`

**Independent**
- `add r1 = r2, r3` (not parallelizable)
- `sub r4 = r11, r2` (not parallelizable)
- `shl r5 = r14, r8` (not parallelizable)

- **Compiler knows the available parallelism**
  - and now HAS the “vocabulary” to express it - STOPs (;;)

- **Hardware easily exploits the parallelism**

**Frees up hardware for parallel execution**
Itanium Architecture Performance Features

**Itanium : Explicitly Parallel**

- **Itanium template specifies**
  - The type of operation for each instruction
    - MFI, MMI, MII, MLI, MIB, MMF, MFB, MMB, MBB, BBB
    - Intra-bundle (M;;MI or MI;;I) and Inter-bundle stop

- **Most common combinations covered by templates**
  - Headroom for additional templates

- **Simplifies hardware requirements**

- **Scales compatibly to future generations**

**Basis for increased parallelism**

- Memory (M)
- Memory (M)
- Integer (I)
- Template (MMI)

128 bits (bundle)
Architecture Challenges

- Sequential Semantics of the ISA
  - Low Instruction Level Parallelism (ILP)
  - Unpredictable Branches, Mem dependencies
  - Ever Increasing Memory Latency
  - Limited Resources (registers, memory addr)
  - Procedure call, Loop pipelining Overhead
  - ...

Itanium EPIC ISA : Sequential--, Parallel++
Predication

Traditional Arch

- cmp
- br

else

then

Itanium

- cmp p1,p2
- p2
- p1
- p2

Control flow to Data flow

Predication removes/reduces branches
Predication ...

- **Unpredictable branches removed**
  - Misprediction penalties eliminated

- **Basic block size increases**
  - Compiler has a larger scope to find ILP

- **ILP within the basic block increases**
  - Both “then” and “else” executed in parallel

- **Wider machines are better utilized**

Predication enables and enhances ILP
Itanium Architecture Performance Features

**Normal Compares**

- **Two kinds of normal compares**
  - Regular
  - Unconditional (nested IF’s)

Regular: p3 is set just once

Unconditional: p3 and p4 are AND’ed with p2

Opportunity for Even More Parallelism...
Introducing Parallel Compares

- Nested conditionals and compound conditionals (&&, ||) frequently require a sequence of conditions
- Three new types of compares:
  - **AND**: if cond=false, sets both predicates FALSE
  - **OR**: if cond=true, sets both predicates TRUE
  - **OR.ANDCM**: if cond=true, sets one TRUE, other FALSE

Reduces Critical Path
Architecture Challenges

- Sequential Semantics of the ISA
- Low Instruction Level Parallelism (ILP)
- Unpredictable Branches, Mem dependencies
- Ever Increasing Memory Latency
- Limited Resources (registers, memory addr)
- Procedure call, Loop pipelining Overhead
- ...

Itanium Predication: ILP++, Branches--
Control Speculation

Itanium Architecture Performance Features

Traditional Arch

```
instr 1
instr 2
br
```

```
ld r1=
use =r1
```

Itanium

```
ld.s r1=
instr 1
instr 2
br
```

```
chk.s r1
use =r1
```

Branch barrier broken! Memory latency addressed

Load moved above branch by compiler
Speculative data uses can also be speculated.

Itanium

Uses moved above branch by compiler

Recovery code

Control speculating “uses” further increases ILP
Introducing the NaT ("Not a Thing")

NaT is the GR’s 65th bit that indicates:
- whether or not an exception has occurred
- branch to fixup code required

- NaT set during ld.s, checked by Chk.s
Itanium Architecture Performance Features

**NaT Propagation**

- All computation instructions propagate NaTs to reduce number of checks required

### Code Example

```
ld8.s r3 = (r9)
ld8.s r4 = (r10)
add r6 = r3, r4
ld8.s r5 = (r6)
p1,p2 = cmp(...)
(p1) br
```

### Code Example

```
ld8 r3 = r(9)
ld8 r4 = (r10)
add r6 = r3, r4
ld8 r5 = (r6)
br home
```

### Code Example

```
chk.s r5, rec
home:
    sub r7 = r5,r2
rec:
    ld8 r3 = r(9)
    ld8 r4 = (r10)
    add r6 = r3, r4
    ld8 r5 = (r6)
    br home
```
Architecture Challenges

- Sequential Semantics of the ISA
- Low Instruction Level Parallelism (ILP)
- Unpredictable Branches, Mem dependencies
- Ever Increasing Memory Latency
- Limited Resources (registers, memory addr)
- Procedure call, Loop pipelining Overhead
- ...

Itanium Control Speculation: ILP++, Latency impact--
Data Speculation

Traditional Architectures

- instr 1
- instr 2
- st [?

Barrier

- ld r1=
- use =r1

Itanium

- ld.a r1=
- instr 1
- instr 2
- st [?

- ld.c r1
- use =r1

Load moved above store by compiler

Store barrier broken! Memory latency addressed
Speculative data uses can be speculated.

- Uses moved above store by compiler.

Data speculating "uses" further increases ILP.
Itanium Architecture Performance Features

Advanced Load Address Table - ALAT

- ld.a inserts entries.
- Conflicting stores remove entries
  - Also: ld.c.clr, chk.a.clr,
- Presence of entry indicates success
  - chk.a branches when no entry is found

ld.a reg# =...
chk.a reg# → ?

reg # | Address
-----|------
reg # | Address
reg # | Address

st
Architecture Challenges

- Sequential Semantics of the ISA
- Low Instruction Level Parallelism (ILP)
- Unpredictable Branches, Memory dependencies
- Ever Increasing Memory Latency
- Limited Resources (registers, memory addr)
- Procedure call, Loop pipelining Overhead
- ...

Itanium Data Speculation: ILP++, Latency impact--
An abundance of machine resources

Itanium Architecture Performance Features

Massive Execution Resources

Integer Registers
- GR0
- GR1
- GR31
- GR32
- GR127

63 0

0

32 Static

96 Stacked, Rotating

NaT

Floating-Point Registers
- FR0
- FR1
- FR31
- FR32
- FR127

81

0

32 Static

96 Rotating

Branch Registers
- BR0
- BR7

63

0

Predicate Registers
- PR0
- PR1
- PR15
- PR16
- PR63

bit 0

16 Static

48 Rotating
Massive Memory Resources

- 18 BILLION Giga Bytes accessible
  - $2^{64} = 18,446,744,073,709,551,616$
- Both 64-bit and 32-bit pointers supported
- Access granularity and alignment
  - 1, 2, 4, 8, 10, 16 bytes
  - Alignment on naturally aligned boundaries is recommended
  - Instructions are 16 byte aligned and little endian ordered
- Both Little and Big Endian Order supported

An abundance of memory resources
Architecture Challenges

- Sequential Semantics of the ISA
- Low Instruction Level Parallelism (ILP)
- Unpredictable Branches, Mem dependencies
- Ever Increasing Memory Latency
- Limited Resources (registers, memory addr)
- Procedure call, Loop pipelining Overhead
- ...

Itanium Resources: Aid “explicit” parallelism
Register Stack

- GR Stack reduces need for save/restore across calls
- Procedure stack frame of programmable size (0 to 96 regs)
- Mechanism implemented by renaming register addresses

Distinct resources reduce overhead
Itanium Architecture Performance Features

Register Stack

Frame overlap eases parameter passing
Register Stack Engine (RSE)

- Automatically saves/restores stack registers without software intervention
  - Provides the illusion of infinite physical registers
  - by mapping to a stack of physical registers in memory
  - Overflow: Alloc needs more registers than available
  - Underflow: Return needs to restore frame saved in memory

- RSE may be designed to utilize unused memory bandwidth to perform register spill and fill operations in the background

RSE eliminates stack management overhead
Architecture Challenges

- Sequential Semantics of the ISA
- Low Instruction Level Parallelism (ILP)
- Unpredictable Branches, Mem dependencies
- Ever Increasing Memory Latency
- Limited Resources (registers, memory addr)
  - Procedure call, Loop pipelining Overhead
- ...

Itanium Reg. Stack: Modular program support
Itanium Architecture Performance Features

Software Pipelining Support

- **High performance loops without code size overhead**
  - No prologue/epilogue
    - Register rotation (rrb)
    - Predication
    - Loop control registers (LC, EC)
    - Loop branches (br.ctop, br.wtop)
  - Especially valuable for integer loops with small trip counts

Whole loop computation in parallel

Itanium Loop support: ILP+++ , Overhead---
SW Pipelined Loop Example

• **DAXPY inner loop**: \( dy[i] = dy[i] + (da * dx[i]) \)
  - 2 loads, 1 fma, 1 store / iteration

• **Machine assumptions**
  - can do 2 loads, 1 store, 1 fma, 1 br / cycle
  - load latency of 2 clocks
  - fma latency of 1 clocks (not realistic, but good for example)
Example: Pipeline

- Each column represents 1 source iteration

load dx, dy

dy + da * dx

store dy
Example Code

.rotf dx[3], dy[3], tmp[2]

mov ar.lc = 3     // #iterations-1
mov ar.ec = 4     // #stages
mov pr.rot = 0x10000
;;
looptop:
    (p16) ldfd dx[0] = [dxsp],8
    (p16) ldfd dy[0] = [dysp],8
    (p18) fma.d tmp[0] = da, dx[2], dy[2]
    (p19) stfd [dydp] = tmp[1],8
    br.ctop looptop
    ;;
Loop Execution

Execution Sequence

Before Initialization
Loop Execution

**Execution Sequence**

- (p16) ldₜₓ
- (p16) ldₜᵧ
- (p18) fma
- (p19) st

**Initialization**

- RRB=0
- LC=3
- EC=4

**Execution Sequence**

- 19: 0 (p19)
- 18: 0 (p18)
- 17: 0
- 16: 1 (p16)
- 63: 0 (p63)

Itanium Developer Seminar - Itanium Architecture Overview (37)
Loop Execution

Execution Sequence

(p16) ld_x
(p16) ld_y
(p18) fma
(p19) st

RRB=0

LC=3  EC=4

Prologue

19: 0  (p19)
18: 0  (p18)
17: 0  (p18)
16: 1  (p16)
63: 0  (p63)
Loop Execution

RRB=-1

16: 1
17: 0
18: 0

(p16) \text{ld}_x
(p16) \text{ld}_y

1

62: 0
63: 1

16: 1
17: 0
18: 0

(p19)
(p18)
(p16)

LC=2 EC=4

Branch 1

Execution Sequence

(p18) fma
(p19) st
(p18) fma
(p19) st
Loop Execution

Execution Sequence

(p16) ld_x
(p16) ld_y
(p18) fma
(p19) st
(p18) fma
(p19) st
(p18) fma
(p19) st

RRB=-2
LC=1
EC=4

Branch 2
Loop Execution

<table>
<thead>
<tr>
<th>RRB</th>
<th>LC</th>
<th>EC</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

Execution Sequence

\[ (p16) \text{ ld}_x \quad (p16) \text{ ld}_y \quad (p18) \text{ fma} \quad (p19) \text{ st} \]

Branch 3
Loop Execution

Execution Sequence

\[
\begin{align*}
(p16) \text{ ld}_x & \quad (p16) \text{ ld}_y & \quad (p18) \text{ fma} & \quad (p19) \text{ st} \\
(p16) \text{ ld}_x & \quad (p16) \text{ ld}_y & \quad (p18) \text{ fma} & \quad (p19) \text{ st} \\
(p16) \text{ ld}_x & \quad (p16) \text{ ld}_y & \quad (p18) \text{ fma} & \quad (p19) \text{ st} \\
(p16) \text{ ld}_x & \quad (p16) \text{ ld}_y & \quad (p18) \text{ fma} & \quad (p19) \text{ st} \\
(p16) \text{ ld}_x & \quad (p16) \text{ ld}_y & \quad (p18) \text{ fma} & \quad (p19) \text{ st}
\end{align*}
\]

Loop Execution

RRB = -4

Branch 4

LC = 0
EC = 3
Loop Execution

Branch 5

RRB = -5

LC = 0   EC = 2

Execution Sequence

(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
(p16) ld_x   (p16) ld_y   (p18) fma   (p19) st
Loop Execution

Execution Sequence:

- (p16) ld_x
- (p16) ld_y
- (p18) fma
- (p19) st
- (p16) ld_x
- (p16) ld_y
- (p18) fma
- (p19) st
- (p16) ld_x
- (p16) ld_y
- (p18) fma
- (p19) st
- (p16) ld_x
- (p16) ld_y
- (p18) fma
- (p19) st
- (p16) ld_x
- (p16) ld_y
- (p18) fma
- (p19) st
- (p16) ld_x
- (p16) ld_y
- (p18) fma
- (p19) st
- (p16) ld_x
- (p16) ld_y
- (p18) fma
- (p19) st

RRB=-6

Branch 6

LC=0   EC=1
Loop Execution

 execution sequence

Branch 7

RRB=-7

LC=0 EC=0

...
Architecture Challenges

- Sequential Semantics of the ISA
- Low Instruction Level Parallelism (ILP)
- Unpredictable Branches, Mem dependencies
- Ever Increasing Memory Latency
- Limited Resources (registers, memory addr)
  - Procedure call, Loop pipelining Overhead
- ...

Itanium Loop support: Perf. w/o code overhead
Floating-Point Architecture

- **Fused Multiply Add Operation**
  - An efficient core computation unit
- **Abundant Register resources**
  - 128 registers (32 static, 96 rotating)
- **High Precision Data computations**
  - 82-bit unified internal format for all data types
- **Software divide/square-root**
  - High throughput achieved via pipelining

*Itanium FP: High performance and high precision*
Itanium Architecture Performance Features

-  

- **Memory Hierarchy Control**
  - Allocation, Flush, Prefetch (Data/Inst.), …

- **Multimedia Support**
  - Semantically compatible with Intel’s MMX™ technology and Streaming SIMD Extension instruction technology

- **Bit/Byte field instructions**
  - Population count, Extract/Deposit, Leading/Trailing zero bytes, …

And the performance features continue ...
Itanium Architecture
Performance Features

- Parallelism - inherent in Itanium architecture
- Frees up hardware for parallel execution
- Predication reduces branches, enables/enhances ILP
- Control Specn breaks branch barrier, increases ILP
- Data Specn breaks data dependences, increases ILP
- Control and Data Specn address memory latency
- Itanium provides abundant machine & mem resources
- Stack/RSE reduces call overhead and management
- Loop support yields performance w/o overhead
- And the performance features continue ...

Beyond traditional RISC capabilities
And YES ...

The Compiler CAN and DOES use these powerful architecture features to

- Enable
- Enhance
- Express
- Exploit

the **Parallelism**
Agenda

- Today’s Architecture Challenges
- Novel Itanium Architecture Features
  - Itanium™ Processor Micro-Arch. Overview
- Examples of Itanium Features in Action
Itanium Processor Micro-architecture Overview

Itanium™ Processor Goals

- **World-class performance on high-end applications**
  - High performance for commercial servers
  - Supercomputer-level floating point for technical workstations
- **Large memory management with 64-bit addressing**
- **Robust support for mission critical environments**
  - Enhanced error correction, detection & containment
- **Full IA-32 instruction set compatibility in hardware**
- **Deliver across broad range of industry requirements**
  - Flexible for a variety of OEM designs and operating systems

Deliver world-class performance and features for servers & workstations and emerging internet applications
Parallel, deep, and dynamic pipeline designed for maximum throughput

**Highlights of the Itanium™ Pipeline**

- **6-Wide EPIC hardware under compiler control**
  - Parallel hardware and control for predication & speculation
  - Efficient mechanism for enabling register stacking & rotation
  - Software-enhanced branch prediction

- **10-stage in-order pipeline designed for:**
  - Single cycle ALU (4 ALUs globally bypassed)
  - Low latency from data cache

- **Dynamic support for run-time optimization**
  - Decoupled front end with prefetch to hide fetch latency
  - Aggressive branch prediction to reduce branch penalty
  - Non-blocking caches, register scoreboard to hide load latency
Itanium™ delivers greater instruction level parallelism than any contemporary processor.
Itanium™ Processor Micro-architecture Overview

Maximizing SW-HW Synergy

Architecture Features programmed by compiler:

- Branch Hints
- Explicit Parallelism
- Register Stack & Rotation
- Predication
- Data & Control Speculation
- Memory Hints

Micro-architecture Features in hardware:

- Fetch
  - Instruction Cache & Branch Predictors
  - Fast, Simple 6-Issue
- Issue
  - Fast, Simple 6-Issue
- Register Handling
  - 128 GR & 128 FR, Register Remap & Stack Engine
- Control
  - Fast, Simple 6-Issue
  - Bypasses & Dependencies
- Parallel Resources
  - 4 Integer + 4 MMX Units
  - 2 FMACs (4 for SSE)
  - 2 LD/ST units
  - 32 entry ALAT
- Speculation Deferral Management
- Memory Subsystem
  - Three levels of cache: L1, L2, L3
Itanium™ Processor Micro-architecture Overview

10 Stage In-Order Core Pipeline

Front End
• Pre-fetch/Fetch of up to 6 instructions/cycle
• Hierarchy of branch predictors
• Decoupling buffer

Execution
• 4 single cycle ALUs, 2 ld/str
• Advanced load control
• Predicate delivery & branch
• Nat/Exception//Retirement

Instruction Delivery
• Dispersal of up to 6 instructions on 9 ports
• Reg. remapping
• Reg. stack engine

Operand Delivery
• Reg read + Bypasses
• Register scoreboard
• Predicated dependencies
Floating Point Features

- Native 82-bit hardware provides support for multiple numeric models
- 2 Extended precision pipelined FMACs deliver 4 EP / DP FLOPs/cycle
- Performance for security and 3-D graphics
  - 2 Additional single-precision FMACs for 8 SP FLOPs/cycle (SIMD)
  - Efficient use of hardware: Integer multiply-add and s/w divide
- Balanced with plenty of operand bandwidth from registers / memory
  - 6 x 82-bit operands
  - 2 stores/clk
  - 128 entry 82-bit RF
  - 2 x 82-bit results
  - 2 DP Ops/clk
  - 4 DP Ops/clk (2 x Fld-pair)

Industry-leading floating point performance
IA-32 Compatibility

- **Itanium™ directly executes IA-32 binary code**
  - Shared caches & execution core increases area efficiency
  - Dynamic scheduler optimizes performance on legacy binaries

- **Seamless Architecture allows full Itanium performance on IA-32 system functions**

**Diagram:**

1. Compatibility Fetch & Decode
2. IA-32 Dynamic Scheduler
3. IA-32 Retirement & Exceptions
4. Shared I-Cache
5. Shared Execution Core

**Full, efficient IA-32 inst. compatibility in HW**
Agenda

- Today’s Architecture Challenges
- Novel Itanium Architecture Features
- Itanium™ Processor Micro-Arch. Overview
- Examples of Itanium Features in Action