
Intel™ Itanium™ Architecture Concepts

Allan Knies

Intel Research, Berkeley

Intel

1

Agenda for this Talk

Itanium architecture history and strategies

Itanium application architecture overview

C → Itanium example

Conclusions and Q & A

Intel

3

Agenda for this Talk

Itanium history and strategies

Itanium Application Architecture Overview

C → Itanium example

Conclusions and Q & A

Intel

5

Objectives for This Talk

Provide background for some Itanium architectural decisions

Describe features of Itanium application architecture

- Provide introduction and overview
- Describe software and performance usage models
- Mention relevant design issues
- Lots of slides left for reference...

Show Itanium architecture feature usage

C → assembly example

Intel

2

Itanium Definition History

Two concurrent 64-bit architecture developments

- IAX at Intel from 1991
 - Conventional 64-bit RISC
- Wideword at HP Labs from 1987
 - Unconventional 64-bit VLIW derivative

Itanium definition started in 1994, 1st product 2001!

- Extensive participation of Intel and HP architects, compiler writers, micro-architects, logic/circuit designers
 - Several customers also participated as definition partners
-

Intel

4

Itanium Philosophy Summary

Move complexity of resource allocation, scheduling, and parallel execution to compiler

Provide features that enable compiler to reschedule programs using advanced features (predication, speculation)

Enable, enhance, express, and exploit parallelism

Intel

6

Itanium Strategies Summary

Expressing Parallelism

- Register state
- Instructions groups and bundles
- Parallel compares

Hiding Memory Latency

- Control speculation
- Data speculation

Reducing Branch Effects

- Predication
- Loop-type branches

Supporting Software Modularity

- Register stack

Intel

7

Itanium Strategies – Expressing Parallelism

Problem: Extracting parallelism is difficult

- Standard architectures limit exploitation of parallelism on in-order implementations
- Even OOO chips require lots of compiler support

Strategy

- Enable wider machines through:
 - large register files (keep values in flight)
 - static dependence specification (reduce HW detection)
 - static resource allocation (reduce scheduling overhead)
- Allow parallelism to be expressed at ISA level

Intel

8

Itanium Strategies Summary

Expressing Parallelism

- Register state
- Instructions groups and bundles
- Parallel compares

Hiding Memory Latency

- Control speculation
- Data speculation

Reducing Branch Effects

- Predication
- Loop-type branches

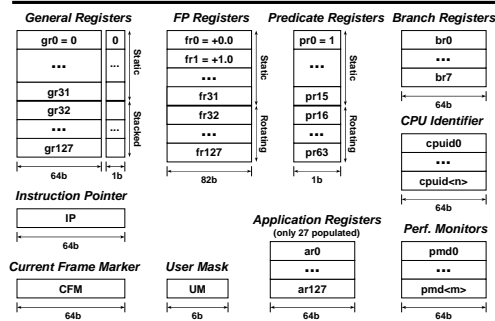
Supporting Software Modularity

- Register stack

Intel

9

Application State



Intel

10

Itanium Strategies Summary

Expressing Parallelism

- Register state
- Instructions groups and bundles
- Parallel compares

Hiding Memory Latency

- Control speculation
- Data speculation

Reducing Branch Effects

- Predication
- Loop-type branches

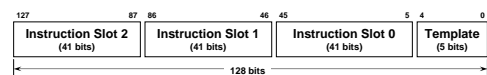
Supporting Software Modularity

- Register stack

Intel

11

Instruction Formats: Bundles



Template identifies types of instructions in bundle and delineates independent operations (through "stops")

Instruction types

- M: Memory
- I: Shifts and multimedia
- A: ALU
- B: Branch
- F: Floating point
- L+X: Long

Template encodes types

- MII, MLX, MMI, MFI, MMF, MI_I, M_MI
- Branch: MIB, MMB, MFB, MBB, BBB

Template encodes parallelism

- All come in two flavors: with and without stop at end

Intel

12

Execution Semantics I

Traditional architectures have sequential semantics

- The machine must always **behave** as if instructions were executed in an unpipelined sequential fashion
- If a machine issues instructions out of order or in parallel, it must insure sequential execution semantics are obeyed

Case 1 – Dependent

```
add r1 = r2, r3
sub r4 = r1, r2
shl r2 = r4, r8
```

Case 2 – Independent

```
add r1 = r2, r3
sub r4 = r11, r21
shl r12 = r14, r8
```

Intel

13

Execution Semantics II

Itanium is an explicitly parallel (EPIC) architecture

- Compiler uses templates with **stops** (";;") to separate "instruction groups"
- Hardware need not check for register dependencies within instruction groups
 - WAR register dependencies generally allowed
 - Memory operations still require sequential semantics
- Predication can disable dependencies dynamically

Case 1 – Dependent

```
add r1 = r2, r3 ;;
sub r4 = r1, r2 ;;
shl r12 = r1, r8
```

3 Instruction groups

Case 2 – Independent

```
add r1 = r2, r3
sub r4 = r11, r21
shl r12 = r14, r8
```

1 Instruction group

Case 3 – Predication

```
(p1) add r1 = r2, r3
(p2) sub r1 = r2, r3 ;;
shl r12 = r1, r8
```

2 Instruction groups
(p1 != p2)

Intel

14

Itanium Strategies Summary

Expressing Parallelism

- Register state
- Instructions groups and bundles
- **Parallel compares**

Hiding Memory Latency

- Control speculation
- Data speculation

Reducing Branch Effects

- Predication
- Loop-type branches

Supporting Software Modularity

- Register stack

Intel

15

Parallel Compares

Parallel compares allow compound conditionals to be placed in a single instruction group and executed in parallel

Example

```
if ( a && b && c ) { . . . }
```

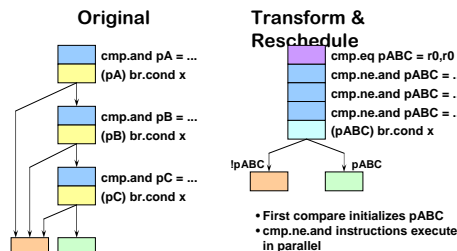
Itanium assembly language

```
cmp.eq p1 = r0, r0 ;; // initialize p1=1
cmp.ne.and p1 = rA, 0
cmp.ne.and p1 = rB, 0
cmp.ne.and p1 = rC, 0
```

Intel

16

Parallel Compares: Height Reduction



Intel

17

Itanium Support for Parallel Compares

Compare

- Equality: `eq, ne`
- Relational (only against zero): `lt, le, gt, ge, etc...`
- Test instructions: `tbit, tnat`

Allows for both 'and' and 'or' compares

- One side: `and`
- One side: `or`
- Both sides of conditional: `or.andcm, and.orcm`

Intel

18

Itanium Strategies Summary

Expressing Parallelism

- Register state
- Instructions groups and bundles
- Parallel compares

Hiding Memory Latency

- Control speculation
- Data speculation

Reducing Branch Effects

- Predication
- Loop-type branches

Supporting Software Modularity

- Register stack

Intel

19

Itanium Strategies – Memory Latency

Problem: Memory latency is difficult to hide

- Increasing relative to processor speed
- Implies larger cache miss penalties

Strategy

- Allow compiler to schedule for longer latencies by using control and data speculation
- Explicit compiler control of data movement through an architecturally visible memory hierarchy
- “Do” and ask forgiveness later...

Intel

20

Speculation

Separates loads into 2 parts: speculative loading of data and conflicts/fault detection...

Control Speculation	Data Speculation
Original: <pre>(p1) br.cond foo ld8 r1 = [r2]</pre>	Original: <pre>st4 [r3] = r7 ld8 r1 = [r2]</pre>
Transformed: <pre>ld8.s r1 = [r2] ... (p1) br.cond foo ... chk.s r1, recover</pre>	Transformed: <pre>ld8.a r1 = [r2] ... st4 [r3] = r7 ... chk.a r1, recover</pre>

Intel

21

Itanium Strategies Summary

Expressing Parallelism

- Register state
- Instructions groups and bundles
- Parallel compares

Hiding Memory Latency

- Control speculation
- Data speculation

Reducing Branch Effects

- Predication
- Loop-type branches

Supporting Software Modularity

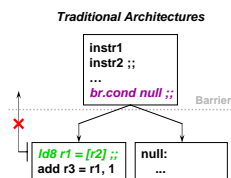
- Register stack

Intel

22

Control Speculation

```
if ( !p ) {
    x = p->val + 1;
    ...
}
```



Control speculation is ...

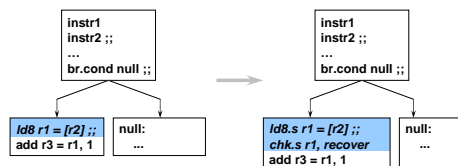
- Moving loads (and possibly instructions that use loaded values) above branches on which their execution is dependent

Three steps...

Intel

23

Control Speculation: Step 1



Separate load behavior from exception behavior

- The `ld.s` which defers exceptions
- The `chk.s` which checks for deferred exceptions

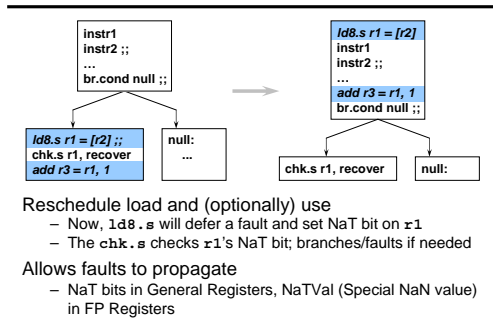
Exception token propagates from `ld.s` to `chk.s`

- NaT bits in General Registers, NaTVal (Special NaN value) in FP Registers

Intel

24

Control Speculation: Step 2



Intel

25

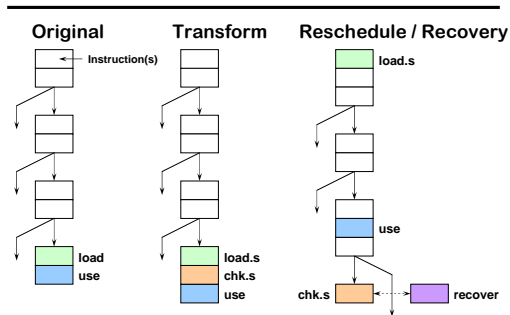
Control Speculation: Step 3



Intel

26

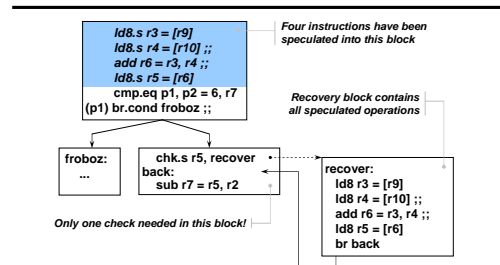
Control Speculation Summary



Intel

27

Loose ends: NaT Propagation



Intel

28

Loose Ends: Faults/Exceptions

Speculation can create speculative loads whose results are never used – want to avoid taking faults/cache misses on these instructions

Deferral allows efficient delay of costly exceptions

NaTs and checks (`chk`) enable deferral with recovery

Exceptions are taken in recovery code, when it is known that result will be used for sure

Intel

29

Itanium Strategies Summary

Expressing Parallelism

- Register state
- Instructions groups and bundles
- Parallel compares

Hiding Memory Latency

- Control speculation
- Data speculation

Reducing Branch Effects

- Predication
- Loop-type branches

Supporting Software Modularity

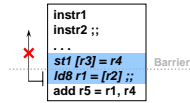
- Register stack

Intel

30

Data Speculation

Traditional Architectures



Data speculation is ...

- Moving loads (and possibly instructions that use loaded values) above potentially overlapping stores

Three steps...

Intel

31

Data Speculation: Step 1



Separate load behavior from overlap detection

- The `ld8.a` performs normal loads and book-keeping
- The `chk.a` checks "the books" to see if a conflict occurred

"The Books": Advanced Load Address Table (ALAT)

Intel

32

Data Speculation: Step 2



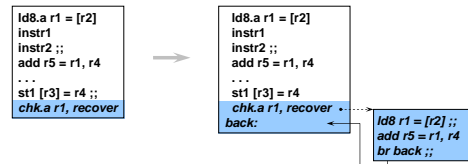
Reschedule load and (optionally) use

- The `ld8.a` allocates an entry in ALAT
- If the `st1` address overlaps with the `ld8.a` address, then ALAT entry is removed
- The `chk.a` checks for matching entry in ALAT - if found, speculation was OK; if not found, must re-execute

Intel

33

Data Speculation: Step 3



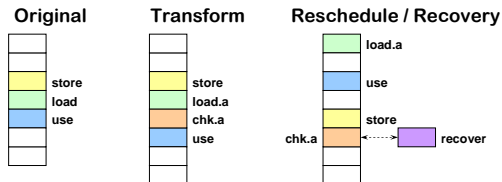
Add recovery code

- All instructions speculated are repeated in a recovery block
- Will only execute if the `chk.a` cannot find ALAT entry
- Use non-speculative versions of instructions

Intel

34

Data Speculation Summary



Intel

35

Itanium Support for Data Speculation

ALAT - Hardware structure that contains information about outstanding advanced loads

Instructions

- Advanced loads: `ld.a`
- Check loads: `ld.c`
- Advance load checks: `chk.a`

Speculative advanced loads - `ld.sa` combines semantics of `ld.a` and `ld.s` (i.e., it is a control-speculative advanced load with fault deferral)

Intel

36

Itanium Strategies Summary

Expressing Parallelism

- Register state
- Instructions groups and bundles
- Parallel compares

Hiding Memory Latency

- Control speculation
- Data speculation

Reducing Branch Effects

- Predication
- Loop-type branches

Supporting Software Modularity

- Register stack

Intel

37

Itanium Strategies -- Branching

Branches interrupt control flow/scheduling

- Mispredictions limit performance
- Even with perfect branch prediction, small basic blocks of code cannot fully utilize wide machines

Strategies

- Allow compiler to eliminate branches (and increase basic block size) with predication
- Allow compiler to schedule more than one branch per clock - multiway branch
- Provide special support for loops
- Reduce number and duration of branch mispredictions by using compiler generated branch hints

Intel

38

Itanium Strategies Summary

Expressing Parallelism

- Register state
- Instructions groups and bundles
- Parallel compares

Hiding Memory Latency

- Control speculation
- Data speculation

Reducing Branch Effects

- **Predication**
- Regular branch types
- Loop-type branches

Supporting Software Modularity

- Register stack

Intel

39

Predication Concepts

Branching causes difficult to handle effects

- Instruction stream changes (reduces fetching efficiency)
- Requires branch prediction hardware
- Requires execution of branch instructions
- Potential branch mispredictions

Itanium provides predication

- Allows some branches to be removed
- Allows some types of safe code motion beyond branches
- Basis for branch architecture and conditional execution

Intel

40

Predication

Using Predicates: Dynamically enable/disable execution

- Most instructions specify a predicate register
- Example: `(p1) add r1 = r2, r3`
 - If (p1) true, add executes normally; else add squashed
- 64 1-bit predicate registers

Generating Predicates: Comparison/test instructions

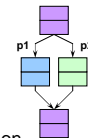
- Most compare/test write result and complement
- Example: `cmp.eq p1, p2 = r1, 0`
 - Performs operation: `p1 ← r1 == 0, p2 ← !(r1 == 0)`

Intel

41

Control Flow Simplification

Original Code



Predicated Code



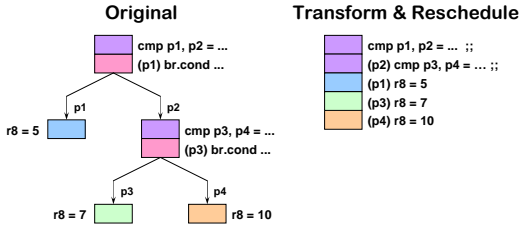
Predication

- Changes control flow dependences into data dependences
- Removes branches
 - Reduce / eliminate mispredictions and branch bubbles
 - Exposes parallelism
 - Increases pressure on instruction fetch unit (but reduces fragmentation)

Intel

42

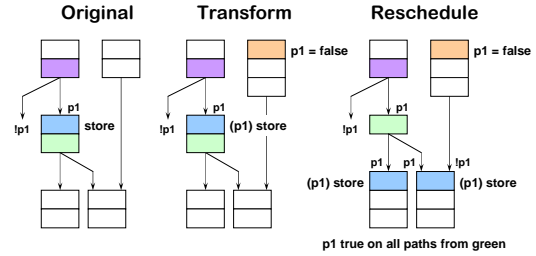
Multiple Selects



Intel

43

Downward Code Motion



Intel

44

Itanium Strategies Summary

Expressing Parallelism

- Register state
- Instructions groups and bundles
- Parallel compares

Hiding Memory Latency

- Control speculation
- Data speculation

Reducing Branch Effects

- Predication
- Regular branches
- Loop-type branches

Supporting Software Modularity

- Register stack

Intel

45

Branch Architecture

Branch registers

- 8 registers for indirect jumps and procedure call/return link

IP-relative branches

- Short form uses 21-bit bundle-aligned displacement
- Long form uses 60-bit bundle-aligned displacement

Multi-way branches

- Group 1 to 3 branches in bundle
- Allow multiple bundles to participate

Loop branches

- Use for "counted" and "while" loops
- Help support software pipelining

Branches have static and dynamic prediction hints

Intel

46

Branch Execution

Examples

```
(p0) br target1 ;; // unconditional

      cmp p1,p2 = <cond> // conditional
(p1) br.cond target2 ;;
```

Compare and branch can be in same instruction group

Intel

47

Multiway Branches

Allow multiple branch targets to be selected in one instruction group

Example

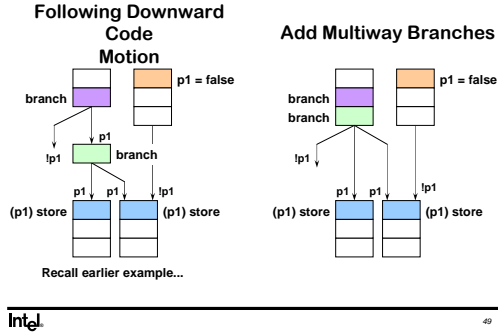
```
(p1) br.cond target_1
(p2) br.cond target_2
(p3) br.call b1
```

Four possible instructions executed next: fall through, target_1, target_2, or address in b1

Intel

48

Multiway Branches: Height Reduction



Intel

49

Itanium Strategies Summary

Expressing Parallelism

- Register state
- Instructions groups and bundles
- Parallel compares

Hiding Memory Latency

- Control speculation
- Data speculation

Reducing Branch Effects

- Predication
- Regular branches
- Loop-type branches

Supporting Software Modularity

- Register stack

Intel

50

Loop Branches

Two counters

- LC counts main loop iterations
- EC counts pipeline stages for drain in pipelined loops

br.cloop

- Uses LC for simple, non-pipelined loops
- Decrements LC and loops until LC is 0

br.ctop, br.wtop, br.cexit, and br.wexit

- Work with register rotation (more on this later...)
- Uses LC, EC for pipelined loops
 - LC, EC control register rotation & predicate initialization
- Details of "counted" and "while" forms follow...

Intel

51

Software Pipelining

```
for (i=0;i<4;i++) { // DAXPY2 loop
    dy[i] = dy[i] + (da * dx[i])
}
```

```
loop: // note stop bits/cycles
    ldff    dx = [dxsp],8 // cycle 1
    ldff    dy = [dysp],8 ;; // cycle 1
    fma.d   tmp = da, dx, dy ;; // cycle 2
    stfd    [dydp] = tmp,8 // cycle 3
    br.cloop loop ;; // cycle 3
```

Overlap execution of different loop iterations (ind iterations)



Intel

52

Software Pipelining: Code

Itanium assembly language

```
.rotf dx[3], dy[3], tmp[2]
    mov ar.lc = 3 // # iterations-1
    mov ar.ec = 4 // # stages
    mov pr.rot = 0x10000
    ;;
loop: // note stop bits/cycles
    (p16) ldff    dx[0] = [dxsp],8
    (p16) ldff    dy[0] = [dysp],8
    (p18) fma.d   tmp[0] = da, dx[2], dy[2]
    (p19) stfd    [dydp] = tmp[1],8
    br.ctop    loop
    ;; // all in cycle 1
```

Intel

53

Software Pipelining: Example

Execution Sequence (iterations)

```
(p16)ldx (p16)ldy (p18)fma (p19)st // iteration 1
(p16)ldx (p16)ldy (p18)fma (p19)st // iteration 2
(p16)ldx (p16)ldy (p18)fma (p19)st // iteration 3
(p16)ldx (p16)ldy (p18)fma (p19)st // iteration 4
(p16)ldx (p16)ldy (p18)fma (p19)st // iteration 5
(p16)ldx (p16)ldy (p18)fma (p19)st // iteration 6
(p16)ldx (p16)ldy (p18)fma (p19)st // iteration 7
```

Serial execution would have been:

4 iterations * 3 cycles/iteration = 12 cycles

What I didn't tell you... rotating registers

Intel

54

Branch Architecture Conclusions

Odds & Ends

- Multiple branches per clock is natural side-effect of speculation
- Allows fast selection of multiple branch targets
- Branch prediction for both single and multiple branches is important for good performance
- Compiler profiling can help facilitate use of hints
- Hints may reduce needed size/functionality of hardware predictors
- Works in conjunction with control speculation, data speculation, predication, rotation, and parallel compares

Intel

55

Itanium Strategies – Support SW Modularity

Procedure calls interrupt scheduling/control flow

- Call overhead from saving/restoring registers
- Software modularity is standard

Strategy

- Reduce procedure call/return overhead
 - Register Stack
 - Register Stack Engine (RSE)

Intel

56

Itanium Strategies Summary

Expressing Parallelism

- Register state
- Instructions groups and bundles
- Parallel compares

Hiding Memory Latency

- Control speculation
- Data speculation

Reducing Branch Effects

- Predication
- Loop-type branches

Supporting Software Modularity

- Register stack

Intel

57

Register Stack

Itanium provides hardware support for stack frames

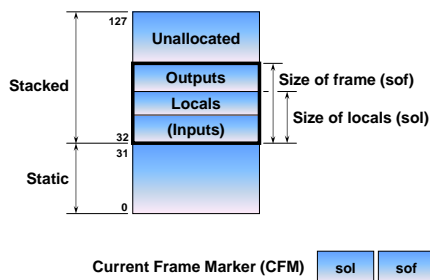
Motivation

- Automatic save and restore of GRs on procedure call and return
- Cache traffic reduction
- Vectorize register spill and fill

Intel

58

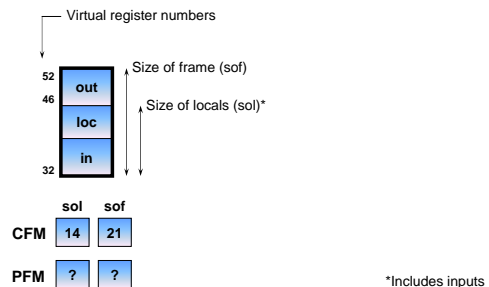
General Registers & Stack Frame



Intel

59

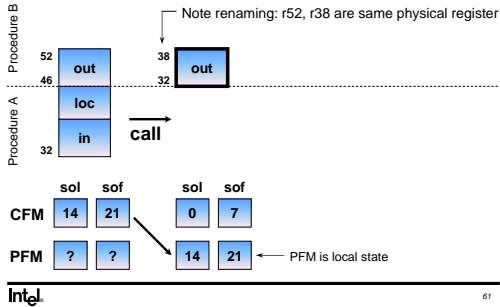
GR Stack Frame: Example



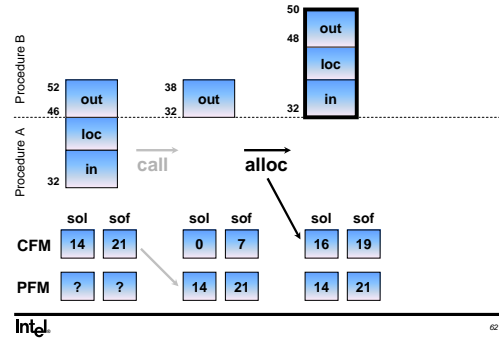
Intel

60

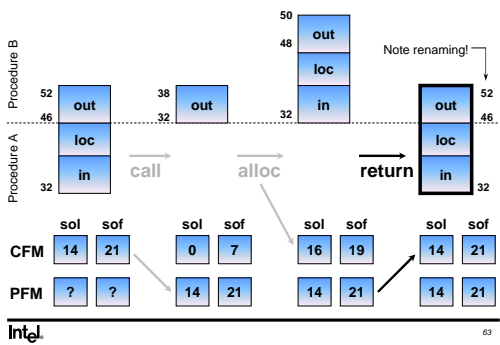
GR Stack Frame: Call



GR Stack Frame: Allocate



GR Stack Frame: Return



Register Stack Engine / Backing Store

Register stack: finite physical depth but infinite virtual depth

- Number of physical registers is ≥ 96

"Overflow"

- When the processor runs out of physical registers (during an alloc), the Register Stack Engine (RSE) automatically spills registers to memory (backing store)

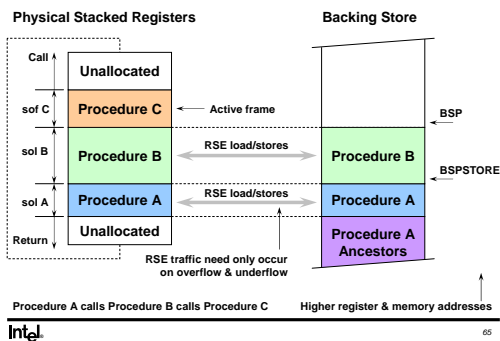
"Underflow"

- When a procedure whose register stack was spilled to backing store returns (br.ret), the RSE restores the registers from the backing store

Thus, programmer does not need to explicitly save or restore registers due to procedure calls (saves code)

Intel

Register Stack Engine: Backing Store



Agenda for this Talk

Itanium history and strategies

Itanium application architecture overview

C → Itanium Example

Conclusions and Q & A

Intel

A C to Itanium Example

Compile `chkGetChunk` function from Spec95 `vortex`

Assumptions for this example

- Abstract machine model (mostly like Itanium Processor)
- Unlimited instruction issue (execution) resources
- Loads have 2 cycle latency to first level cache
- All other instructions have 1 cycle latency
- Elide speculation recovery code

Hold on...

Intel

67

Synthesis: `chkGetChunk` Code

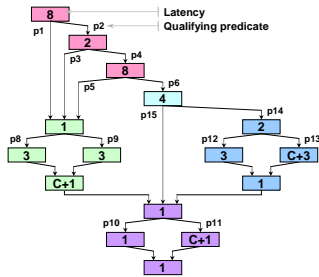
```

if (((Theory->Flags[ChunkNum] & 0x0008)
    && ((Theory->Flags[ChunkNum] & 0x0040)
    && (*(Theory->ChunkAddr[ChunkNum] - 28)) == SizeOfUnit) {
StackPtr = *(Theory->ChunkAddr[ChunkNum] - 20);
if (Index >= StackPtr) {
if (SetGetSwl) {
*Status = -1009;
} else {
Mem_DumpChunkChunk (0, ChunkNum);
*Status = 1005; }
} else {
if (*(Theory->ChunkAddr[ChunkNum] - 28) != SizeOfUnit) {
*Status = 1003;
} else {
*Status = 1004; }
Mem_DumpChunkChunk (0, ChunkNum);
}
}
return((Test = *Status==0 ? True:Ut_PrintErr (F,Z,*Status)););
    
```

Intel

68

Synthesis: Initial CFG



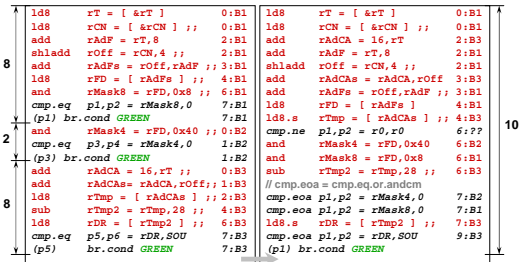
= Cycle count, C = Call latency

$25 \leq \text{Cycles} \leq 2C + 31$

Intel

69

Synthesis: Red Blocks



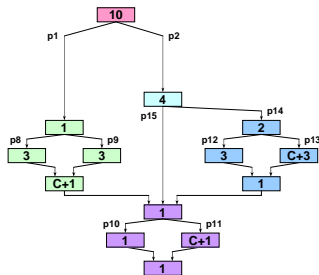
Steps:

- 1) Speculate the `ld8` instructions, reschedule, use parallel compares

Intel

70

Synthesis: Updated CFG

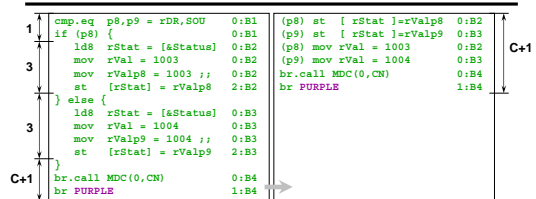


= Cycle count, C = Call latency

Intel

71

Synthesis: Green Blocks



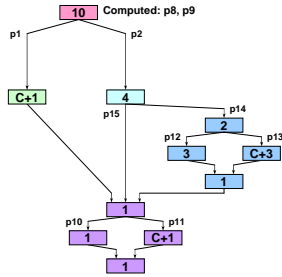
Steps:

- 1) Speculate `cmp.eq` instruction into red block's cycle 9
- 2) Speculate `ld8` instructions into red block's cycle 1
- 3) Copy and speculate `mov` instructions into red block's cycle 0
- 4) Predicate both sides of conditional

Intel

72

Synthesis: Updated CFG

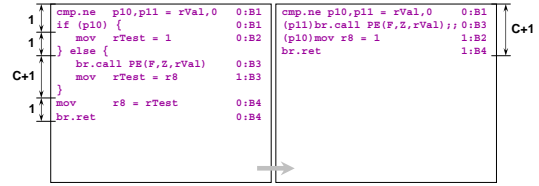


= Cycle count, C = Call latency



73

Synthesis: Purple Blocks



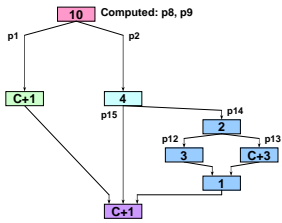
Steps:

- 1) Replace `rTest` with `r8`
- 2) Predicate both sides of conditional



74

Synthesis: Updated CFG

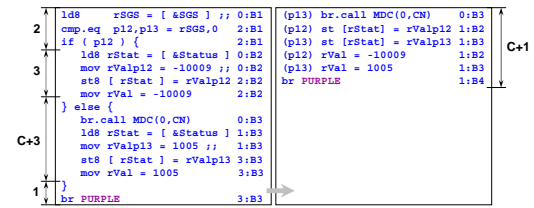


= Cycle count, C = Call latency



75

Synthesis: Blue Blocks



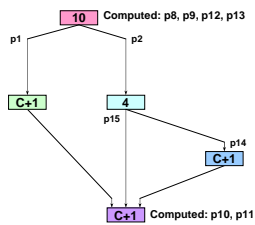
Steps:

- 1) Speculate all `ld8` instructions into red block's cycle 1
- 2) Speculate `cmp.eq` instruction into red block's cycle 3
- 3) Copy and speculate `mov rVal = 1005` instruction into red block's cycle 0
- 4) Predicate both sides of conditional



76

Synthesis: Updated CFG

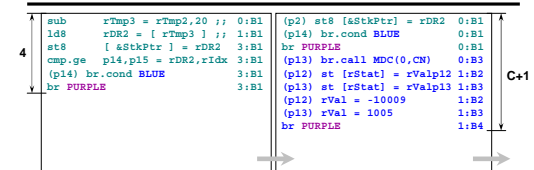


= Cycle count, C = Call latency



77

Synthesis: Turquoise Block I



Steps:

- 1) Speculate `sub`, `ld8`, and `cmp.ge` into red block (in cycles 6, 7, and 9, respectively)
- 2) Predicate `st8` with `p2`
- 3) Concatenate with blue block

Continues on next slide...



78

Application State

Directly accessible CPU state	
- 128 x 65-bit General registers	(GR)
- 128 x 82-bit Floating-point registers	(FR)
- 64 x 1-bit Predicate registers	(PR)
- 8 x 64-bit Branch registers	(BR)
Indirectly accessible CPU state	
- Current Frame Marker	(CFM)
• RRB, *, SOR, SOL, SOF	
- Instruction Pointer	(IP)
Control and Status registers	
- User Mask	(UM)
- 19 Application registers	(AR)
- ≥ 5 CPU Identifiers	(CPUID)
- ≥ 4 Performance monitor data	(PMD)
Memory	

Intel

85

Instruction Interactions with RSE

- On `br.call...`
- Set PFS.pfm to CFM
 - Set sof to sof - sol
 - Set sol, sor, and rrb to 0
 - Create new frame with only output registers
- On `alloc...`
- Resize current frame by setting sof to $i + l + o$, and sol to $i + l$
 - Save PFS to a GR
- On `mov to PFS...`
- Restore PFS from a GR
- On `br.ret...`
- Restore CFM from PFS.pfm
 - Restore local registers for previous frame

Intel

86

Itanium Application Architecture Overview

Application state
 Instruction format and execution semantics
 Integer, floating point, and multimedia instructions
 Memory and semaphores
 Control speculation and data speculation
 Predication
 Parallel compares
 Branch architecture

Register Rotation and Software Pipelining

Register stack

Intel

87

Register Rotation

Motivation

- Pipeline-schedule loops onto hardware
- Remove extraneous work from loop
- Minimize start-up overhead
- Small code footprint
- Maximum computational throughput with few instructions

Intel

88

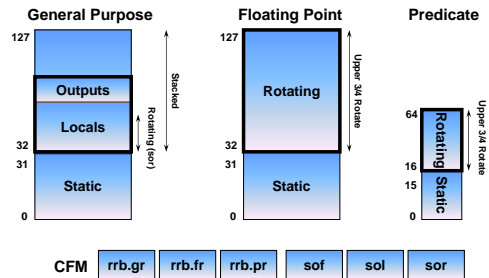
Register Rotation: Odds & Ends

- Rotating Register Base (RRB)
- Separate for GR, FR, and PR register files
 - Loop branches decrement all RRBs
- Instructions contain a "virtual" register number
- $RRB + \text{virtual register number} = \text{physical register number}$
- For GRs
- Size of rotating region is zero or multiple of 8
 - Rotating region overlays current frame
 - Allows rotation & stack renaming in one level of address
 - Region always starts at r32
 - Must copy input registers before loop

Intel

89

Register Rotation: Register Files



Intel

90

Counted and While Loop Branches

br.ctop, br.wtop

- Loop decision located at bottom of loop
- Taken branch continues loop; not-taken branch exits loop
- Behavior
 - br.ctop is taken if $LC \neq 0 \mid \mid EC > 1$, else not-taken
 - br.wtop is taken if $qp == 1 \mid \mid EC > 1$, else not-taken

br.cexit, br.wexit

- Loop decision located other than at bottom of loop
- Not-taken branch continues loop; taken branch exits loop
- Behavior
 - br.cexit is not-taken iff $LC \neq 0 \mid \mid EC > 1$, else taken
 - br.wexit is not-taken iff $qp == 1 \mid \mid EC > 1$, else taken

Intel

91

Software Pipelining

Overlap execution of different loop iterations



Synergistic use of Itanium features

- Predication, special branches
- Register rotation: removes loop copy overhead
- Predicate rotation: removes prologue and epilogue

Traditional architectures use loop unrolling

- High overhead: extra code for loop body, prologue, epilogue
- Especially useful for integer loops with small trip counts

Intel

92

Software Pipelining Example

DAXPY inner loop

- Performs: $dy[i] = dy[i] + (da * dx[i])$
- Each iteration: 2 loads, 1 FP multiply-accumulate, 1 store

Machine assumptions

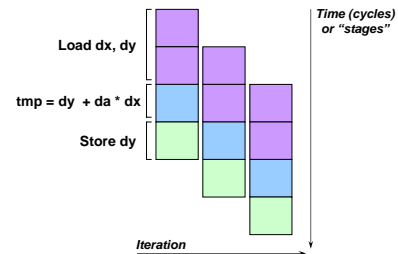
- Can issue 2 loads, 1 store, 1 FMA, 1 branch each cycle
- 2 cycle load latency
- 1 cycle FMA latency (not realistic, but good for example)

Intel

93

Software Pipelining: Pipeline

Each column represents 1 source iteration



Intel

94

Software Pipelining: Code

Itanium assembly language

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

Intel

95

Software Pipelining: Example 1

Initialization: LC = 3, EC = 4, p16 = 1

Predicate Registers Execution Sequence (iterations)

Phys. ID	Vir. ID
19	0 p19
18	0 p18
17	0 p17
16	1 p16
63	0 p63
...	...
rrb.pr	0

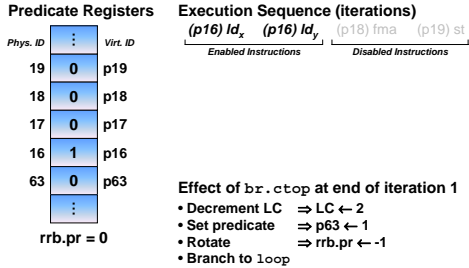
Effect of br.ctop

Intel

96

Software Pipelining: Example II

LC = 3, EC = 4: Load [0]...

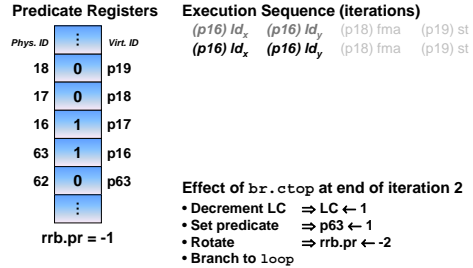


Intel

97

Software Pipelining: Example III

LC = 2, EC = 4: Load [1]...

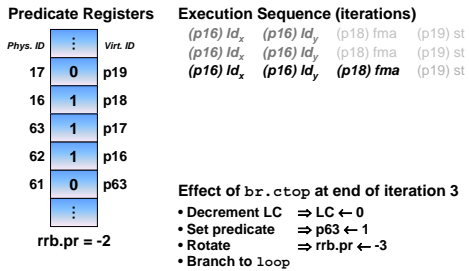


Intel

98

Software Pipelining: Example IV

LC = 1, EC 4: Load [2], compute [0]...

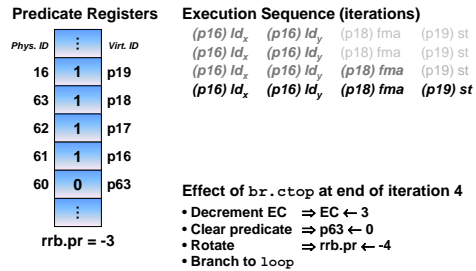


Intel

99

Software Pipelining: Example V

LC = 0, EC = 4: Load [3], compute [1], store[0]...

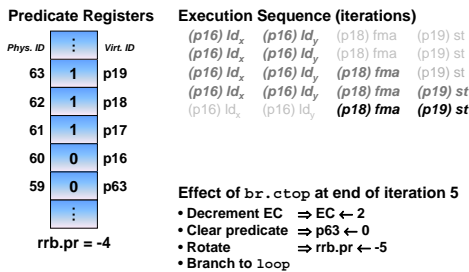


Intel

100

Software Pipelining: Example VI

LC = 0, EC = 3: Compute [2], store[1]...

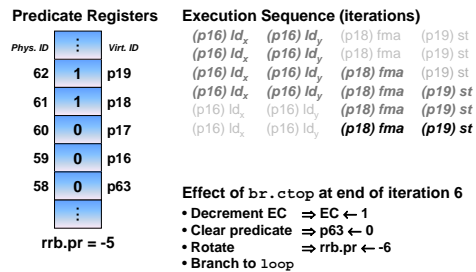


Intel

101

Software Pipelining: Example VII

LC = 0, EC = 2: Compute [3], store[2]...



Intel

102

Software Pipelining: Example VIII

LC = 0, EC = 1: Store[3]...

Predicate Registers		Execution Sequence (iterations)			
Phys. ID	Virt. ID	(p16) ld _x	(p16) ld _y	(p18) fma	(p19) st
61	1	(p16) ld _x	(p16) ld _y	(p18) fma	(p19) st
60	0	(p16) ld _x	(p16) ld _y	(p18) fma	(p19) st
59	0	(p16) ld _x	(p16) ld _y	(p18) fma	(p19) st
58	0	(p16) ld _x	(p16) ld _y	(p18) fma	(p19) st
57	0	(p16) ld _x	(p16) ld _y	(p18) fma	(p19) st
	...				
	rrb.pr = -6				

Effect of `br.ctop` at end of iteration 7

- Decrement EC ⇒ EC ← 0
- Clear predicate ⇒ p63 ← 0
- Rotate ⇒ rrb.pr ← -7
- Fall through

Intel

103

Software Pipelining: Example IX

LC = 0, EC = 0: State after loop

Predicate Registers		Execution Sequence (iterations)			
Phys. ID	Virt. ID	(p16) ld _x	(p16) ld _y	(p18) fma	(p19) st
60	0	(p16) ld _x	(p16) ld _y	(p18) fma	(p19) st
59	0	(p16) ld _x	(p16) ld _y	(p18) fma	(p19) st
58	0	(p16) ld _x	(p16) ld _y	(p18) fma	(p19) st
57	0	(p16) ld _x	(p16) ld _y	(p18) fma	(p19) st
56	0	(p16) ld _x	(p16) ld _y	(p18) fma	(p19) st
	...				
	rrb.pr = -7				

Fall through...

Intel

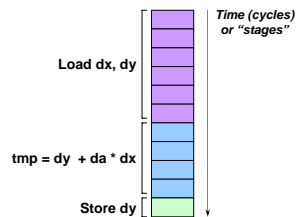
104

Software Pipelining: New Pipeline

Suppose latencies change

- Load latency becomes 6 clocks
- FMA latency becomes 4 clocks

An iteration now looks like this...



Intel

105

Software Pipelining: New Code

New Itanium assembly language (type denotes changes)

```
.rotf dx[7], dy[7], tmp[5]
    mov ar.lc = 3 // # iterations-1
    mov ar.ec = 11 // # stages
    mov pr.rot = 0x10000
    ;;
loop:
    (p16) ldfd      dx[0] = [dxsp], 8
    (p16) ldfd      dy[0] = [dysp], 8
    (p22) fma.d     tmp[0] = da, dx[6], dy[6]
    (p26) stfd      [dydp] = tmp[4], 8
    br.ctop        loop
    ;;
```

Intel

106

Software Pipelining: Summary

Software pipelining maximizes performance; minimizes overhead

- Avoids code expansion of unrolling and code explosion of prologue and epilogue
- Smaller code means fewer cache misses
- Greater performance improvements in higher latency conditions

Reduced overhead allows software pipelining of small loops with unknown trip counts

- Typical of integer scalar codes

Intel

107

Performance Monitoring

Intel

108

Why Monitor Performance?

Alternative is trace-driven simulation

- Slow
- Difficult

Measurements made using performance monitors drive

- Compiler design and optimizations
- 64-bit O/S and application development
- Design of future processors

Performance monitoring is critical for understanding

- Compiler optimizations
- Micro-architectural structures (caches, ALAT, TLB, etc.)

Primary usage models

- Workload Characterization
- Profiling

Intel

109

Workload Characterization

Goals

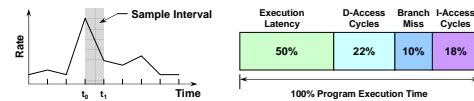
- Understand behavior and characteristics of workload
- Determine if performance problems exist

Event Rate Monitoring

- Time-Based Sampling: IPC, Cache miss/sec, TLB miss/sec
- Event-Based Sampling: Correlation of IPC & events

Cycle Accounting

- Classify each CPU cycle into 8 measurable categories and/or 4 derivable categories



Intel

110

Profiling

Goals

- Understand which code locations cause performance losses
- Identify and fix performance bottlenecks
- Optimize data placement & instr. scheduling in critical loops
- Guide use of speculation & predication by compiler

Instruction Pointer Sampling

- Time/event-based uses architectural IP
- Number of instrs. in pipe limits accuracy (54 in Itanium™)

Event Address Registers (EARs)

- Expose "event address" to software (I and D addresses)
- Enable software to associate events with code/data locations
- Statistical "event address" sampling has finer resolution; enables profile-guided compilation & detailed application tuning

Intel

111

Performance Monitoring Register Set

Two sets of registers

- Architectural
- Implementation-specific extensions

Register types

- 4 performance counters (PMC/PMD pairs)
- 2 opcode match registers
- Instruction and data Event Address Registers (EAR)
- Branch trace buffer
- Instruction and data address range check registers
- Interrupt status

Intel

112

Event Rate Monitoring

Clock cycle counter (ITC register)

Retired instruction counter

50+ Micro-architectural model-specific events

Event occurrence and duration counters

Multi-occurrence counters with thresholding capability

Simple event counting

Time interrupt for time-based counter sampling

Event count overflow interrupt for event-based sampling

Intel

113

More on Events

Single occurrence events

- Event types limited to rate of one per cycle (e.g. TLB misses)
- Duration counters that count the number of cycles during which a condition persists

Multiple occurrence events

Overlapping concurrent events

Intel

114

Cycle Accounting

Stall Counters

- Determine program's cycle break-down by stall reason; accounts for overlapped pipeline stalls

CPU prioritizes overlapping stalls in reverse order

- Report stalls that occur later in pipe and that overlap with earlier stage stalls as being caused by later stage

Measurable stall events

- Combined: branch, memory, execution, and instruction fetch
- Branch mispredict, instruction access, data access, and execution latency

Derivable stall events

- RSE activity, issue limit, taken branch, and fetch window

Intel

115

Event Qualification

Types of qualification

- Instruction opcode match (Itanium code only)
- Instruction address range check (Itanium code only)
- Data address range check (Itanium memory operations only)
- Event-specific unit masks
- Privilege level check
- Instruction set (i.e., IA-32 or Itanium)
- Performance Monitor Freeze

Address range checks

- Use IBR and DBR; monitored code cannot also use!
- Base address and address range must be power of 2

Intel

117

Software Divide Example I

Scaling: a common operation

Two-dimensional Hydro-dynamics kernel

- Livermore FORTRAN Kernel #18
- Several independent iterations contain divide...

```

DO 70 k= 2,KN
DO 70 j= 2,JN
ZA(j,k) = (ZP(j-1,k+1)+ZQ(j-1,k+1) - ZP(j-1,k) - ZQ(j-1,k))
          * (ZR(j,k)+ZR(j-1,k)) / (ZM(j-1,k)+ZM(j-1,k+1))
ZB(j,k) = (ZP(j-1,k)+ZQ(j-1,k) - ZP(j,k) - ZQ(j,k))
          * (ZR(j,k)+ZR(j,k-1)) / (ZM(j,k)+ZM(j-1,k))
70 CONTINUE
    
```

$$ZA_i = N_i/D_i$$

Intel

119

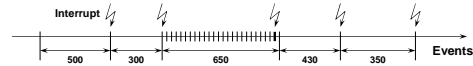
Event Address Registers

IP Sampling using architectural IP is inaccurate

- Number of instructions in flight limits accuracy and resolution (Itanium™ can have about 54 instructions in flight)

Event Address Registers (EARs)

- Hardware captures "event address" that caused "miss event"
- An "Event address" has finer resolution/granularity than IP
- Event capture is countable event
- Statistical "event address" sampling interrupts after observing a programmable number of events



Intel

116

Performance Monitoring Summary

Extensive event set

- Allows continuous "event rate" monitoring: IPC, cache miss rates, etc

Hardware support for real-time cycle accounting

- Total execution cycles broken into four categories

Accurate statistical sampling of miss events

- Event address registers & branch trace buffer support sampling branch mispredicts, cache misses, TLB misses, etc

Capability to "zoom-in"

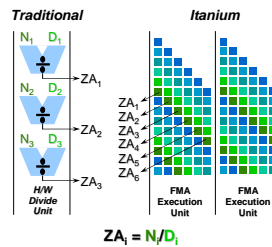
- Event qualification can examine performance of a particular DLL, function, code sequence, or data structure

Intel

118

Software Divide Example II

Itanium software divide provides much greater throughput on FP loops



- Software divide breaks a divide op into several pipelined FMA ops
- Slightly greater latency of each divide, but much greater throughput
- Performance scales as CPU becomes wider and has more FMA units

Intel

120

Addressing Modes

Itanium provides 2 addressing modes

- Base register and auto-increment
- Base register

add, shladd

- Used for all 64b address computations

addp4, shladdp4

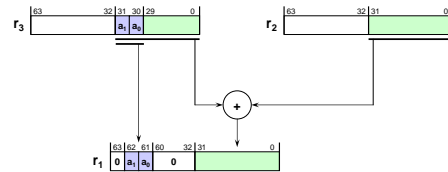
- Used for 32b address computations in 32b Itanium apps
- Modeless 32b support; 64b O/S can seamlessly support 32b and 64b apps
- Behavior (example follows...)
- "Swizzle" copies $r_3\{31:30\}$ into $r_1\{62:61\}$
- Zero extends $r_1\{63,60:32\}$ to remove undefined bits
- Spreads 4GB virtual space into lower 2^{30} bytes of regions 0 to 3 - regions 4 to 7 are reserved for 64b O/S ports

Intel

121

32b Addressing Modes: Example

Example: **addp4** $r_1 = r_2, r_3$

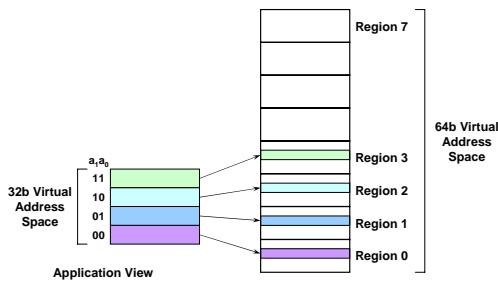


shladdp4 similar; but shifts r_2 left before adding to r_3

Intel

122

32b Swizzled Virtual Addresses



Intel

123

Visibility Definitions

Visibility is architecturally-visible effects of performing an instruction:

- Load \Rightarrow access hits cache or non-visible snoop (in a local sense) structure (e.g., a store buffer)
- Store \Rightarrow access enters snoop (in an MP sense) structure
- UC/WC reference \Rightarrow access reaches the bus

Intel

124

Instruction Ordering Semantics

Release semantics

- `cmpxchg.rel, fetchadd.rel, st.rel, ptc.g, ptc.ga`

Acquire semantics

- `cmpxchg.acq, fetchadd.acq, xchg, ld.acq`, and `ld.c.c1r.acq` that misses the ALAT
- No strong flavors of `ld.s`, `ld.sa`, `ld.c` (what would be the point?)

Fence semantics

- `mf`

Unordered semantics

- `ld, st, ld.s, ld.a, ld.sa, lfetch, ld8, fill, st8, fill, mf.a, fwb, sync.i, ld.c` that misses ALAT

Intel

125

Memory Ordering Executions

Examples

- Relaxed ordering model
- Enforcing basic ordering
- Loads may pass stores
- Preventing loads from passing stores
- Dependencies do not establish MP ordering
- Store buffers may satisfy local loads
- Preventing store buffer satisfaction
- Total order of release and semaphore operations
- Causality

For more discussion consult

- *Itanium Architecture Software Developer's Manual, Volume 2* (Chapter 13, specifically...)

Intel

126

Relaxed Memory Ordering Model

Processor #0			Processor #1		
st	[x] = 1	// M1	ld	r1 = [y]	// M3
st	[y] = 1	// M2	ld	r2 = [x]	// M4

Itanium allows all outcomes

Intel 127

Enforcing Basic Ordering

Processor #0			Processor #1		
st	[x] = 1	// M1	ld.acq	r1 = [y]	// M3
st.rel	[y] = 1	// M2	st	r2 = [x]	// M4

Itanium only disallows outcome r1 = 1, r2 = 0

Intel 128

Enforcing Basic Ordering

Processor #0			Processor #1		
st	[x] = 1	// M1	ld.acq	r1 = [y]	// M3
st.rel	[y] = 1	// M2	st	r2 = [x]	// M4

Itanium only disallows outcome r1 = 1, r2 = 0

Intel 129

Loads may Pass Stores

Processor #0			Processor #1		
st.rel	[x] = 1	// M1	st.rel	[y] = 1	// M3
ld.acq	r1 = [y]	// M2	ld.acq	r2 = [x]	// M4

Itanium allows all outcomes
Note that `ld` and `st` reference different locations

Intel 130

Preventing Loads from Passing Stores

Processor #0			Processor #1		
st.rel	[x] = 1	// M1	st.rel	[y] = 1	// M4
mf		// M2	mf		// M5
ld.acq	r1 = [y]	// M3	ld.acq	r2 = [x]	// M6

Itanium only disallows outcome r1 = 0, r2 = 0
Note that `ld` and `st` reference different locations
`mf` forces M1 \Rightarrow M3 and M4 \Rightarrow M5

Intel 131

Dependencies Don't Establish MP Order

Processor #0			Processor #1		
st	[x] = 1 ; ;	// M1	ld.acq	r2 = [y]	// M4
ld	r1 = [x] ; ;	// M2	ld	r3 = [x]	// M5
st	[y] = r1 ; ;	// M3			

Itanium allows outcome r1 = 1, r2 = 1, r3 = 0
Data dependencies *do not* establish an MP ordering!
– They only order local operations!

Intel 132

Store Buffers may Satisfy Local Loads

Processor #0			Processor #1		
st.rel	[x] = 1	// M1	st.rel	[y] = 1	// M4
ld.acq	r1 = [x]	// M2	ld.acq	r3 = [y]	// M5
ld	r2 = [y]	// M3	ld	r4 = [x]	// M6

Itanium allows outcome r1 = 1, r3 = 1, r2 = 0, r4 = 0

Intel

133

Preventing Store Buffer Satisfaction

Processor #0			Processor #1		
st.rel	[x] = 1	// M1	st.rel	[y] = 1	// M5
mf		// M2	mf		// M6
ld.acq	r1 = [x]	// M3	ld.acq	r3 = [y]	// M7
ld	r2 = [y]	// M4	ld	r4 = [x]	// M8

Itanium disallows outcome r1 = 1, r3 = 1, r2 = 0, r4 = 0

Intel

134

“Total Order”

Processor #0	Processor #1	Processor #2	Processor #3
st.rel [x] = 1 // M1	ld.acq r1 = [x] // M2 ld r2 = [y] // M3	st.rel [y] = 1 // M4	ld.acq r3 = [y] // M5 st r4 = [x] // M6

Itanium disallows outcome r1 = 1, r3 = 1, r2 = 0, r4 = 0
However, Itanium *allows* if M1 or M4 is weak store
– Semaphores and release stores are seen by all observers in a single “total order”

Intel

135

Causality

Processor #0	Processor #1	Processor #2
st.rel [x] = 1 // M1	ld.acq r1 = [x] // M2 st [y] = 1 // M3	ld.acq r2 = [y] // M4 ld r3 = [x] // M5

M2 and M3 could be “causally-related”
Itanium disallows outcome r1 = 1, r2 = 1, r3 = 0
However, Itanium *allows* if M1 is weak store

Intel

136

D.S. Optimizations: Load Hoisting I

Removing loads from loops, consider...

```
while (cond) {
    c = *a + *b; // probably loop invariant
    *ptr++ = c; // ptr may point to a or b
}
```

Remove computation from loop using advanced loads:

```
ld4.a r1 = [ a ]
ld4.a r2 = [ b ] ;;
add r3 = r1, r2
while (cond) {
    chk.a.nc r1, recover1
L1: chk.a.nc r2, recover2
L2: *p++ = r3
}
```

Intel

137

D.S. Optimizations: Load Hoisting II

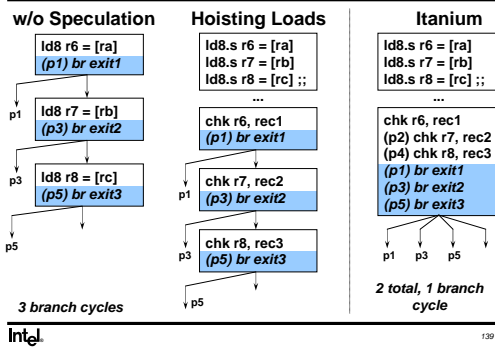
Recovery code:

```
recover1:
    ld4.a r1 = [ a ] ;;
    add r3 = r1, r2
    br.sptk L1
recover2:
    ld4.a r2 = [ b ] ;;
    add r3 = r1, r2
    br.sptk L2
```

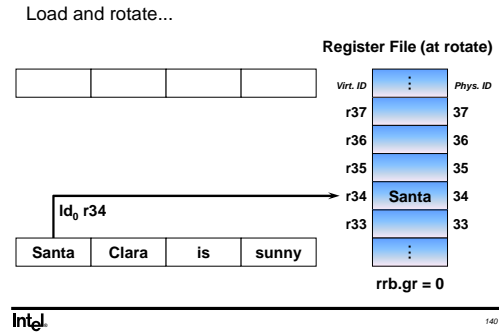
Intel

138

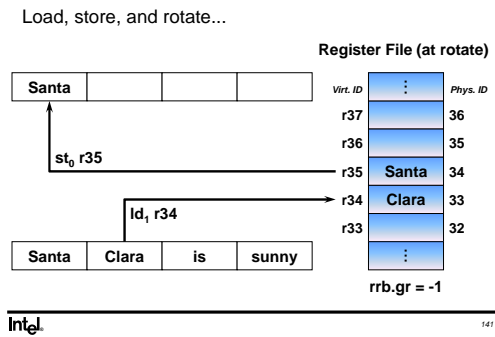
Control Height Reduction



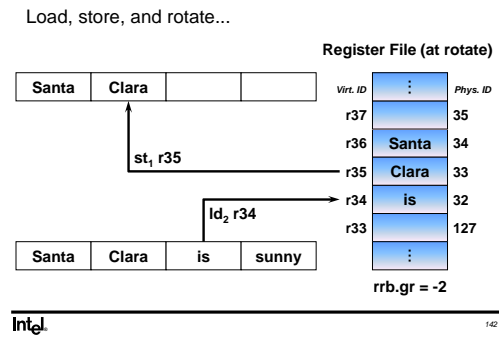
Register Rotation: Example I



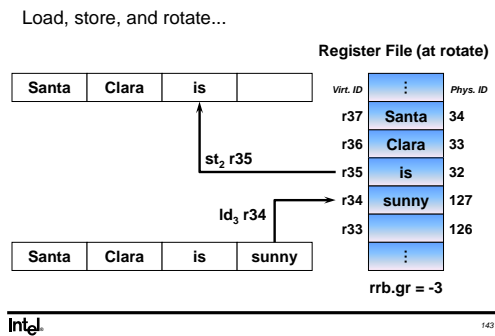
Register Rotation: Example II



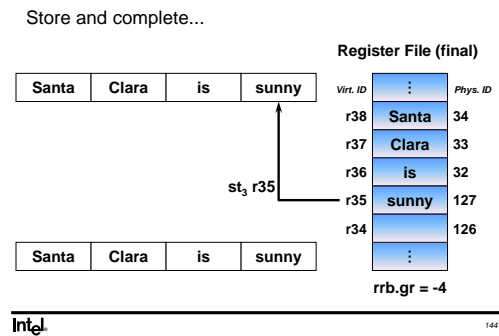
Register Rotation: Example III



Register Rotation: Example IV



Register Rotation: Example V



Itanium Application Architecture Overview

Application state
Instruction format and execution semantics

Integer, Floating Point, and Multimedia Instructions

Memory and semaphores
Control speculation and data speculation
Predication
Parallel compares
Branch architecture
Register rotation and software pipelining
Register stack

Intel

145

Integer Instructions

Memory – load, store, semaphore, ...
Arithmetic – add, subtract, **shladd**, ...
Compare – lt, gt, eq, ne, ..., **tbit**, **tnat**
Logical – and, and complement, or, exclusive-or
Bit fields – deposit, extract
Shift pair
Character
Shifts – left, right, constant and variable
32-bit support – **cmp4**, **addp4**, **shladdp4**, ...
Move – various register files moves
No-ops

Intel

146

Floating-point Architecture

IEEE 754 compliant
Single, double, and double extended (80-bit) formats
– Canonical representation in 82-bit FP registers
128 floating-point registers
– Rotating, not stacked
Fused multiply-add instruction
Load double/single pair instructions
Multiple FP status registers for speculation

Intel

147

Parallel FP Support

Enable cost-effective 3D Graphics platforms
Treat floating-point registers as 2 32-bit single-precision elements
– Full IEEE compliance
• Single, double, double-extended data types, packed-64
– Similar instructions as for scalar floating-point
– Allows fast non-IEEE divide
Exploit data parallelism in applications using 32-bit floating-point data
– Most application's geometry calculations (transforms and lighting) are done with 32-bit floating-point numbers
– Provides 2X increase in computation resources for 32-bit data parallel floating-point operations

Intel

148

Multimedia Support

Audio and video functions typically perform same operation on arrays of data values
Itanium provides instructions that treat general registers as 8 x 8, 4 x 16, or 2 x 32-bit elements; three types
– Addition and subtraction (including special purpose forms)
– Left shift, signed right shift, and unsigned right shift
– Pack/unpack to convert between different element sizes
Semantically compatible with IA-32's MMX Technology™

Intel

149

Itanium Application Architecture Overview

Application state
Instruction format and execution semantics
Integer, floating point, and multimedia instructions

Memory and Semaphores

Control speculation and data speculation
Predication
Parallel compares
Branch architecture
Register rotation and software pipelining
Register stack

Intel

150

Memory

Byte addressable, accessed with 64-bit pointers

Two addressing modes

- Base register
- Base register and auto-increment
- Synthesize others with `add`, `shladd`, `addp4`, `shladdp4`

Access granularity and alignment

- 1, 2, 4, 8, 10, 16 bytes
- Instructions are always 16-byte aligned
- Supports big- or little-endian byte order for data accesses
- For best performance, align on natural boundaries

Ordered and unordered memory access instructions

- Model is relaxed
- Ordering applies to cacheable, write-back memory types

Intel

151

How Itanium Orders Operations

Memory ordering semantics define relationship between operations to *different* addresses

- All observers honor these relationships (for cacheable data)

Dependencies define relationship between operations that have register, memory, or control dependencies

- Itanium resolves memory RAW, WAW, WAR in program order
- Do not *necessarily* establish an MP ordering

Aligned *release stores*, *release & acquire semaphores* become visible to all observers in a single **total order**

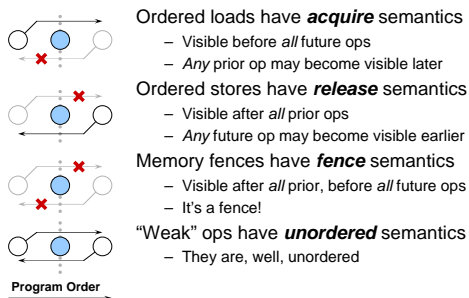
Non-programmer-visible state may satisfy local read requests before data becomes globally visible

- Store buffers, processor caches, etc.

Intel

152

Memory Ordering Semantics



Intel

153

Ordering versus Dependency

Dependency \Rightarrow same address (data-flow too)

Ordering semantics \Rightarrow different address

Both impose an "order" on two memory operations

- But they are **not** interchangeable concepts!

Dependency	Ordering	Ordering
<pre>st [x] = r1 ld r2 = [x]</pre>	<pre>st [x] = r1 ld r2 = [y]</pre>	<pre>st [x] = r1 mf ld r2 = [x]</pre>

Dependence or ordering model may prevent motion

Hardware performs re-order (e.g., non-blocking cache)

Intel

154

Cache Coherency

Cache coherency

- All processor caches maintain consistency with other processors within a coherency domain based on physical addresses
- Instruction caches are not required to maintain coherency with store traffic (for Itanium)
- Software must ensure self-modifying code (using processor stores) is correctly handled through architecturally-defined sequence
- DMA writes guaranteed to snoop instruction and data caches

Intel

155

Semaphores

Atomic read-modify-write operations are performed in cache only with exclusive cache line ownership

- Non-cacheable memory or unaligned semaphores fault
- All semaphores are atomic

xchg.acq

- Exchange register and memory

cmpxchg.acq|rel

- Exchange register and memory if memory is equal to a value
- Application register (AR.ccv) specifies compare value

fetchadd.acq|rel

- Increment memory by small immediate
- Effective in MPP "Take a number" scheduling algorithms
- Implementations may export to memory controllers

Intel

156

Memory Hierarchy Control

Explicit control of cache allocation and deallocation

- Specify levels of memory hierarchy affected by access
- Allocation and flush resolution is at least 32-bytes

Allocation

- Allocation hints indicate at which level allocation takes place
- Used in load, store, and explicit prefetch instructions

Deallocation and flush

- Invalidates the addressed line in all levels of cache hierarchy
- Write data back to memory if necessary

Intel

157

Register Stack: Explicit Management

flushrs

- Flushes all frames except for the active frame to Backing Store
- Stalls execution if necessary

loadrs

- Loads programmable amount of memory below BSP into physical registers
- Stalls execution if necessary
- Amount of memory to be loaded is set in RSC.loadrs

cover

- Creates a new frame with size zero (sof = sol = 0) above current
- Saves CFM in IFM
- Return from interrupt, `rfi`, restores CFM from IFM

Intel

158

Register Stack Engine: RSE Control

RSC: Register Stack Configuration Register (AR16)

Mode hints

- Lazy or synchronous operation
- Load intensive or load asynchronous
- Store intensive or store asynchronous
- Eager or asynchronous

Privilege level, endian mode

- RSE operation depends on data in register stack
- Not on the mode in PSR

Tearpoint distance (`loadrs`)

Intel

159

Register Stack Engine: State

Registers

- 96 x 65-bit virtual registers
- At least 96 physical registers
- Memory backing store

Backing Store Pointer, BSP

- Points to backing store address of GR32 of current frame

BSPStore

- Points to memory location for next RSE store

RNAT

- RSE NaT collection register (64-bits)

RSC

- RSE control register

Intel

160

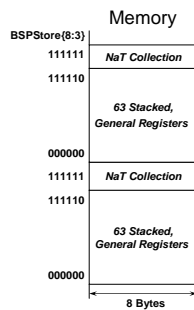
Register Stack Engine: NaT bits

RSE responsible for saving NaT bits

- Spilled/filled in groups of 63
- Every 63 register saves/restores
- NaTs are collected in RNAT register

When `BSPStore(8:3) == 111111`

- RNAT register is stored
- Bit 0 of RNAT corresponds to earliest stored register
- Bit 62 of RNAT corresponds to last stored register
- Bit 63 is always zero



Intel

161