Intel® Itanium™ Processor Reference Manual for Software Optimization

November 2001

Document Number: 245473-003

Contents

1.0 Revision History

Date of Revision	Revision Number	Description
March 2000	1.0	Initial release of document.
August 2000	2.0	Second revision.
November 2001	3.0	Third revision. Document title change from *Itanium Processor Microarchitecture Reference* to *Itanium Processor Reference Manual for Software Optimization.*

2.0 Overview

The Itanium™ processor is the first implementation of the Itanium architecture. This document describes features of the Itanium processor's implementation of the Itanium architecture which are relevant to performance tuning, compilation, and assembly language programming. Unless otherwise stated, all of the restrictions, rules, sizes, and capacities described in this chapter apply specifically to the Itanium processor and may not apply to other implementations.

A general understanding of processor behavior and an explicit familiarity with Itanium instructions are assumed (Please see the *Intel® Itanium Architecture Software Developer's Manual*, Document Number 245317 for Volume 1, Document Number 245318 for Volume 2, and Document Number 245319 for Volume 3 for a full description of the Itanium architecture).

This document describes a wide variety of topics related to the Itanium processor from a software perspective. The details provided are intended for the performance programmer or compiler writer. Note that the amount of space devoted to each topic is not necessarily a representation of how important the topic is. For example, predication, control speculation, data speculation, functional unit resources and latencies are all likely to have a greater effect on performance than using the branch hints even though a large amount of space has been devoted to describing the branch hints.

The information provided here is as accurate as can be described in a short, concise description of the behavior of the Itanium processor microarchitecture. Since the goal is to provide information for performance tuning of broad range of applications, some information (especially related to virtual memory, control registers, branch and prefetch, and deeper levels of the memory pipeline) is approximate because full description of the behavior is beyond the scope of this document.

3.0 Function Units and Issue Rules

This section describes the number and type of functional units available, rules that will avoid unnecessary execution stalls, and topics related to instruction placement and issue.

3.1 Execution Model

The Itanium processor issues and executes instructions in software-supplied order, so compiler understanding of stall conditions is essential for generating efficient assembly code.

In general, when an instruction does not issue at the same time as the instruction immediately before it, instruction execution is said to have *split issue*. When a split issue condition occurs, all instructions after the split point stall one or more clocks, even if there are sufficient resources for some of them to execute. Split issue refers to a stall of this sort even though the stall could be caused by scoreboarding, instruction alignment, functional unit oversubscription, or other pipeline-related stalls.

Common causes of split issue in the Itanium processor are:

- There are insufficient resources to execute an instruction.
- Any of the source registers for an instruction are not yet available from their producing instructions.
- A stop is encountered.
- Instructions have not been placed in accordance with issue rules on the Itanium processor.
- Instruction group alignment rules have not been followed.

3.2 Functional Unit Class Definitions

The table below classifies Itanium instructions by *instruction classes* that are used to define instruction latencies, bypasses, and functional unit capabilities. The classes are provided here only as a notation for describing latencies and resources on the Itanium processor and are not part of the Itanium architectural specification.

Functional Unit Class Name	List of Instructions
BR	br.call, br.cond, br.ia, brl.call, brl.cond, br.ret, br.wexit
BR_B2	br.cexit, br.cloop, br.ctop, br.wexit, br.wtop
BRP	brp,brp.ret
CHK_ALAT	chk.a.clr, chk.a.nc
CHK_I	chk.s.i
CHK_M	chk.s.m
CLD	ld.c
FCLD	ldf8.c, ldfd.c, ldfe.c,l dfp8.c, ldfpd.c, ldfps.c, ldfs.c
FCMP	fclass.m, fcmp
FCVTFX	fcvt.fx, fcvt.fxu, fcvt.xf
FLD	ldf8, ldf8.a, ldf8.s, ldf8.sa, ldfd, ldfd.a, ldfd.s, ldfd.sa, ldfe, ldfe.a, ldfe.s, ldfe.sa, ldf.fill, ldfs, ldfs.a, ldfs.s, ldfs.sa
FLDP	ldfp8, ldfp8.a, ldfp8.s, ldfp8.sa, ldfpd, ldfpd.a, ldfpd.s, ldfpd.sa, ldfps, ldfps.a, ldfps.s, ldfps.sa
FMAC	fma, fnma
FMISC	famax, famin, fand, fandcm, fmax, fmerge.ns, fmerge.s, fmerge.se, fmin, fmix, for, fpack, frcpa, frsqrta, fselect, fswap, fsxt, fxor
FOTHER	fchkf, fclrf, fsetc
FRAR_I	mov.i =ar
FRAR_M	mov.m = ar
FRBR	mov =br
FRCR	mov =cr
FRFR	getf

Functional Unit Class Name	List of Instructions
FRIP	mov =ip
FRPR	mov =pr
IALU	add, addl, adds, shladd, sub
ICMP	cmp4,cmp
ILOG	and, andcm, or, xor
ISHF	dep, dep.z, extr, shrp
LD	ld, ld.a, ld.s, ld.sa, ld.bias
LFETCH	lfetch
LONG_I	movl
MMALU_A	padd, padd4, pavg1, pavg2, pavgsub, pcmp, pshladd2, pshradd2, psub
MMALU_I	pmax, pmin, psad1
MMMUL	pmpy2, pmpyshr2, popcnt
MMSHF	mix, mux, pack, pshl, pshr, shl, shr, unpack
NOP_B	break.b, nop.b
NOP_I	break.i, nop.i
NOP_M	break.m, nop.m
NOP_F	break.f, nop.f
NOP_X	break.x, nop.x
PNT	addp4, shladdp4
RSE_B	clrrrb, cover
RSE_M	flushrs, loadrs
SEM	cmpxchg, fetchadd, xchg
SFCVTFX	fpcvt.fx, fpcvt.fxu
SFMAC	fpma, fpms, fpnma
SFMERGESE	fpmerge.se
SFMISC	fpamax, fpamin, fpcmp, fpmax, fpmerge.ns, fpmerge.s, fpmin, fprcpa, fprsqrta
STF	stf8, stfd, stfe, stfs, stf.spill
ST	st, st8.spill
SYST_B2	bsw,rfi
SYST_B	epc
SYST_M0	alloc, cc, fc, halt, itc.d, itc.i, itr.d, itr.i, mf.a, probe, ptc.e, ptc.g, ptc.ga, ptc.l, ptr.d, ptr.i, rsm, rum, ssm, sum, tak, thash, tpa, ttag, mov psr=, mov =psr, mov rr=, mov =rr, mov pkr=, mov =pkr, mov pmd=, mov =pmd, mov pmc=, mov =pmc, mov msr=, mov =msr, mov ibr=, mov =ibr, mov dbr=, mov =dbr, mov =cpuid
SYST_M	fwb, invala, invala.e, mf, srlz.d, srlz.i, sync.i
TBIT	tbit
TOAR_I	mov.i ar=
TOAR_M	mov.m ar=
TOBR	mov br=
TOCR	mov cr=
TOFR	setf
TOPR	mov pr=
XMA	xma, xmpy
XTD	czx, sxt, zxt

3.3 Number and Types of Functional Units

While Itanium instruction groups may extend over an arbitrary number of bundles and contain an arbitrary number of each instruction type, the Itanium processor has finite execution resources. If an instruction group contains more instructions than there are execution units for that type of instruction, the first instruction for which an appropriate unit cannot be found will cause a split issue.

Execution units are usually classified by slot type: I, F, M, or B. A-type instructions in the Itanium Architecture can be scheduled to execute on either M or I type units. Note that the functional units on the Itanium processor are generally asymmetric with respect to the set of instructions they can execute. For example, even though there are two I-units, I0 and I1, only I0 can execute the `tbit` instruction.

All of the computational functional units are fully pipelined so each functional unit can accept one new instruction per clock cycle in the absence of other types of stalls. There are some exceptions for access to system instructions and registers, but such issues are beyond the scope of this document.

The next sections describe the functional units and the type, number, and classes of instructions that each can execute.

3.3.1 Integer

Integer ALUs execute I-type and A-type instructions. The Itanium processor has two integer (I) units that function as follows:

- I0 can execute all I-type and A-type instructions.
- I1 can execute all I-type and A-type instruction classes except: SYS_I0, FRIP, FRBR, TOBR, MMMUL, TBIT, ISHF, TOPR, FRPR, TOAR_I, FRAR_I.

3.3.2 Memory

Memory units execute M-type and A-type instructions. The Itanium processor has two memory (M) units that function as follows:

- M0 can execute all M-type and A-type instructions.
- M1 can execute all M-type and A-type instruction classes except: SEM, FRFR, SYST_M0, RSE_M, TOAR_M, FRAR_M, TOCR, FRCR.

3.3.3 Branch

Branch units execute B-type instructions. The Itanium processor has three branch (B) units that function as follows:

- B0 and B2 can consume all B-type instructions. However, B2 cannot consume SYST_B0
- B1 can execute all B-type instructions except SYST_B2; however, `brp` instructions sent to B1 will be ignored.

3.3.4 Floating-point

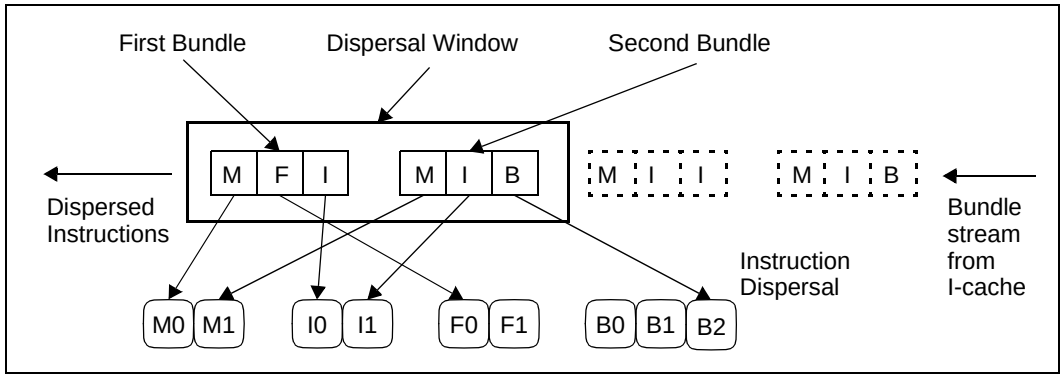
Floating-point execution units execute F-type instructions. The Itanium processor has two floating-point (F) units that function as follows:

- F0 can execute all F-type instructions.
- F1 can execute all F-type instruction classes except: FMISC, SFMISC, FCMP, SFMERGESE.

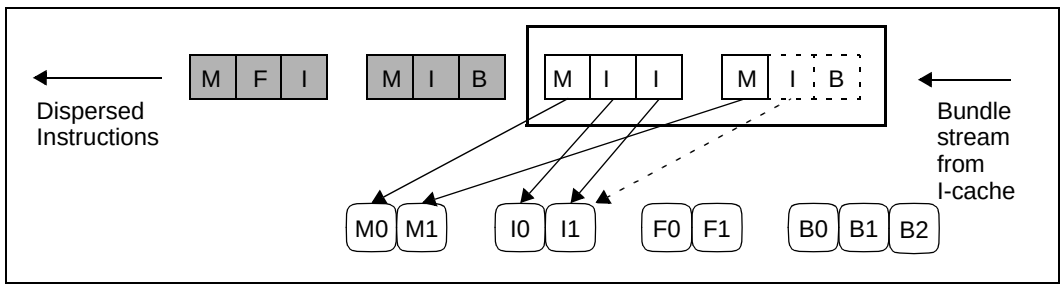
3.4 Instruction Slot to Functional Unit Mapping

The specific Itanium processor functional unit to which an instruction is sent is determined by its instruction slot type and its position within the current set of instructions being issued. The process of sending instructions to functional units is called *dispersal*. The Itanium processor hardware makes no attempt to reorder instructions to avoid stalls. Thus, the compiler must be careful about the number, type, and order of instructions inside an instruction group to avoid unnecessary stalls. The presence of predicates on instructions has no effect on dispersal – all instructions are dispersed in the same fashion whether predicated true, predicated false, or unpredicated. Similarly, nops are dispersed to functional units as if they were normal instructions.

When deciding on functional units for instruction dispersal, the Itanium processor views at most two bundles at a time. This text refers to these bundles as the *first* and *second* bundles. A *bundle rotation* causes new bundles to be brought into the two-bundle window of instructions being considered for issue.

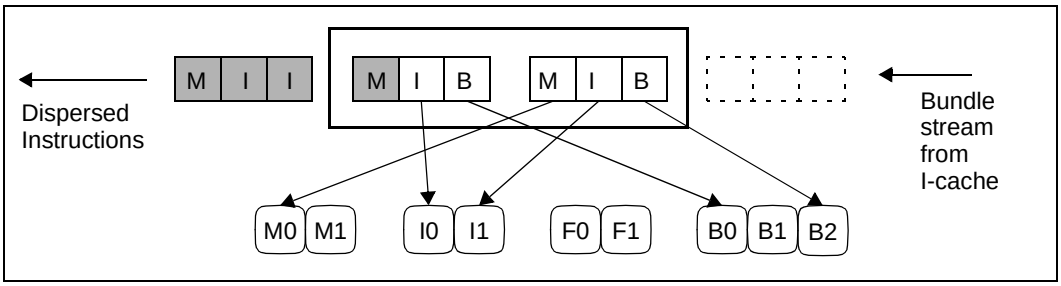


If all of the instructions in both the first and second bundle are issued as shown above, a two bundle rotation occurs: The rotation brings in new first and second bundles as shown below:



In the state shown above, all of the instructions in the first bundle have been issued, but not all of the instructions in the second have been issued. The reason the last two instructions did not issue is

because there were no I units left for the second slot of the second bundle. This will cause a single bundle rotation will be performed as shown next:



The instructions in the first bundle have continued down the execution pipeline, the second bundle has become the first, and a new second bundle has been rotated in. The following rules describe the effect of split issue on bundle rotation:

- Any split issue condition in the first bundle prevents bundle rotation from occurring.
- In the absence of split issue conditions in the first bundle, any split issue condition in the second bundle will cause a single bundle rotation to occur.
- If there are no split issues in the dispersal window, a double bundle rotation results.

If the dispersal rules would cause an instruction to be issued to a functional unit that does not have resources to execute that instruction (for example, if a `tbit` were dispersed to I1), or all functional units capable of executing that instruction have already been taken by earlier instructions, a resource oversubscription occurs and issue splits before the instruction.

If some of the instructions in the dispersal window are not dispatched, the dispersal rules are retried in the next cycle after the bundle rotation rules are applied. This approach guarantees forward progress because after a split issue, some of the instructions earlier in the group have already issued and are no longer competing for issue resources.

3.4.1 Execution Width

Since the Itanium processor looks at two bundles at a time during dispersal, it can issue a maximum of six instruction slots per clock. This six slot collection can contain at most two I-slots, two M-slots, two F-slots, and three B-slots regardless of the instructions in those slots and the values of any predicates.

3.4.2 Dispersal Rules

The dispersal rules for execution units vary according to slot type type and are summarized below:

Rules for M/I slots	An I slot in the third position of the second bundle is always dispersed to I1.
	Otherwise, an M or I instruction is dispersed to the lowest numbered M or I unit not already in use.
Rules for F slots	An F slot in the first bundle disperses to F0.
	An F slot in the second bundle disperses to F1.
Rules for B slots	Each B slot in an MBB or BBB bundle disperses to the corresponding B unit. That is, a B slot in the first position of the template is dispersed to

B0. In the second position it is dispersed to B1. In the third position it is dispersed to B2.

The B in an MIB/MFB/MMB bundle disperses to B0 if it is a `brp` or `nop.b`, otherwise it disperses to B2.

Rules for L slots — An MLX bundle uses ports equivalent to an MFI bundle.

3.4.3 Instruction Dispersal Examples

Since the Itanium processor has one constant width integer shifter, the following sequence will split issue after the `add`, and no bundle rotation will occur. The `extr` and all the instructions in the second bundle will issue in the next cycle, and a double bundle rotation will follow:

```
{ .mii
   ld4  r1=[r5] // Maps to M0, first cycle
   add  r2=r5,r6// Maps to I0, first cycle
   extr r3=r8,5,3// Stall - extr on I0 only
         // extr Issues in second cycle to I0.
}
{ .mbb
   ld4    r20=[r22]
(p2)br.cond L1
(p1)br.cond L2
}
```

However, if the compiler reorders the instructions so that `extr` comes first, then all six instructions issue in one clock, followed by a double bundle rotation:

```
{ .mii
   ld4    r1=[r5]   // Maps to M0
   extr   r3=r8,r5,3// Maps to I0
   add    r2=r5,r6  // Maps to I1
}
{ .mbb
   ld4    r20=[r22] // Maps to M1
   br.cond L1       // Maps to B1
(p1)br.cond L2      // Maps to B2
}
```

3.4.4 Split Issue and Bundle Types on the Itanium™ Processor

Only certain pairs of bundle templates can be used together without causing split issue conditions. This due to the fact that all slots are issued to functional units even if they contain nops. For example, independent of the particular instructions residing in the slots, each of the following sequences of bundle pairs will split issue due to functional unit oversubscription:

MMI **M**MI — Splits issue before third M slot because M0 and M1 are both busy.

MII M**I**I — Splits issue before third I because both I0 and I1 are busy.

MMI **M**II — Splits issue before third M as M0 and M1 are busy.

MII MF**I** — Splits issue before third I as I0 and I1 are both busy.

In addition to resource oversubscription, there are several Itanium processor-specific special cases based on slot types and instructions that will cause split issue:

MMF	Always splits issue before first M and after F regardless of surrounding bundles and stops.
BBB/MBB	Always splits issue after either of these bundles.
MIB/MFB/MMB	Splits issue after any of these bundles unless the B slot contains a `nop.b` or a `brp` instruction.
MIB BBB	Splits issue after the first bundle in this pair.

Note that all of the split issue conditions described here are implementation dependent and explicit stops must still be used to delimit instruction groups to generate legal Itanium-based code.

There are a few other special split issue conditions. The Itanium processor will split issue after `mf.a`, `halt.mf`, `inval`, `invala.e`, and after any instruction in class SEM

3.5 Instruction Group Alignment

If an instruction group extends across a cache-line boundary in the first level instruction cache, there are intricate rules that determine if the instruction group will split issue at the boundary. As a general rule, these types of stalls are likely to occur early in a basic block. Thus, a compiler mainly needs to consider instruction group alignment at the beginning of basic blocks in which the first instruction group crosses a cache line.

Another alignment issue is related to the dispersal window: Since the dispersal window on the Itanium processor is exactly two bundles wide and only moves in whole-bundle increments, any instruction group that spans three or more bundles is guaranteed to split issue at the end of the second bundle even if no resources are oversubscribed.

The example below shows an instruction group that spans more than two bundles. The effective issue times are to the right of each instruction:

```
L:
{ .mii
   add     r1=r2,r3 // 0
   add     r4=r5,r6 ;;// 0
   sub     r7=r8,r9 // 1
}
{ .mfi
   ld4     r14=[r56]// 1
   fadd    f10=f12,f13// 1
   add     r16=r18,r19 // 1: Split issue occurs
                       //    after this instr
}
{ .mmi
   st4     [r16]=r67 ;; // 2
   add     r24=r56,r57  // 3
   add     r28=r58,r59  // 3
}
```

In this sequence, the Itanium processor has sufficient functional units to execute the instruction group starting at the `sub` instruction and ending with the `st4` instruction in one cycle; however, execution will split issue at the end of the second bundle since the instruction group extends beyond the dispersal window.

4.0 Itanium™ Processor Latencies and Bypasses

An instruction *I* scheduled in clock *i* has a total latency of *N* cycles to instruction *J* if *J* can be scheduled no sooner than cycle *i+N* to avoid stalls due to latency. There are two components that determine the total latency of pairs of instructions:

- The latency of the producing instruction.
- The time to bypass the result to the dependent operation.

The next section provides the total latency for most common cases. Some of the most important exceptions to these latencies are also described in Section 4.4. There are a very small number of highly uncommon and irregular cases that are not covered by these tables.

On the Itanium processor, instructions whose operands are not ready (operands are being written, but have not yet finished being written) will stall until its operands are ready. For example, in the code below:

```
ld4    r1=[r5] ;;
add    r2=r1,r6
```

The add instruction will stall until the result of the load is written (2 or more cycles on the Itanium processor). Similarly, in the case where two instructions in separate instruction groups are writing the same register (if they were in the same instruction group, the code sequence would be architecturally undefined), the second writer will wait for the first writer to complete:

```
ld4    r1=[r5] ;;
add    r1=r2,r6
```

Thus, whether the second instruction is reading or writing the register in question, the second instruction will wait for the first instruction to finish writing its result before proceeding. In the case above, if the ld4 misses in cache, the add instruction will have to wait until the value returns from memory before it can proceed.

4.1 Important Functional Unit Total Latencies

The table below describes the latencies of broad classes of operations, but there are a significant number of exceptions to these rules. The full instruction latency table has several asymmetric cases and requires consideration of the type of instruction producing the result, the functional unit on which that instruction is executing, and the type of the consumer instruction. The classes provided here are only used as a notation for describing latencies on the Itanium processor and are not part of the Itanium architectural specification.

Overall, these latencies are correct for producer-consumer pairs in which both instructions execute on the same type of unit: multimedia (MMxxx classes), floating point (Fxxx), parallel floating point (SFxxx), integer (IALU, ILOG, ICMP, LD, ST):

Source Instruction Class	General Description	Latency (Cycles)
FCMP	Floating compare to branch	1
	Floating compare to non-branch instruction	2
FCVTFX	Convert to fix	7
FMAC	Floating arithmetic	5

Source Instruction Class	General Description	Latency (Cycles)
FMISC	Floating min, max, frcpa, . . .	5
FRAR_M, FRAR_I	mov =ar.xx (M/I slot instruction)	See Section 7.0
FRCR	mov =cr (register dependent)	See Section 7.0
FRFR	FP to GP register copy	2
FRIP	mov =ip	2
FRPR	mov =pr	2
IALU	Integer ALU	1
ICMP, TBIT	Integer compare to dependent branch	0
	Integer compare	1
ILOG	Logical	1
ISHF	dep, extr, shrp	1
MMALU_A	A type multimedia	2
LONG	Long mov	1
MMALU_I	I type multimedia.	2
MMMUL	Parallel multiply	2
MMSHF	shl, shr, unpack, pshl, pshr,. . . .	2
PNT	Pointer adds/shladd	1
SEM	semaphore operations	See Section 7.0
SFCVTFX	SIMD fcvtfx	7
SFMAC	SIMD FMACs	5
SFMERGESE	SIMD fmerge.se	7
SFMISC	Miscellaneous SIMD FP	5
SYST	(reg dependent latency)	See Section 7.0
TOAR_I, TOAR_M	I/M-type mov ar.xx= (reg dependent latency)	See Section 7.0
TOCR	mov cr= (reg dependent latency)	See Section 7.0
TOFR	GP to FP register copy	9
TOPR	mov pr=	1
XMA	FP integer multiply (to another XMA)	7
XTD	sxt, zxt, czx	1

4.2 Memory System Latencies and Penalties

The following latencies are for memory operations and memory related flushes.

Source Instruction Class or Event	Description	Latency (Cycles)
CHK_I, CHK_M, CHK_ALAT	chk.a (ALAT hit), chk.s (no NaT/NatVal)	0
	chk.a (ALAT miss), chk.s (NaT/NatVal)	50+
CLD, FCLD	ld*.c (ALAT hit, L1/L2 hit)	0
	ld*.c (ALAT miss, L1/L2 hit)	10
FLD, FLDP	FP load (L2 hit)	9
	FP load (L3 hit)	24

Source Instruction Class or Event	Description	Latency (Cycles)
LD	Integer loads except ld.c (L1 hit)	2
	Integer loads except ld.c (L2 hit)	6
	Integer loads except ld.c (L3 hit)	21
DTC Miss	Number of bubble cycles in pipeline due to DTC miss flush	10

4.3 Branch Related Latencies and Penalties

The following latencies are for branch operations and branch related flushes.

Source Instruction Class or Event	Description	Latency (Cycles)
FRBR	mov =br	2
TOBR	mov br=	1 to non-branches 0 to BRs
Mispredicted Branch Penalty	Dead cycles from time branch is executed to time next instruction is started	9
Taken Branch Bubbles	Number of front end bubbles inserted on a correctly predicted taken branch	0/1/2/3

4.4 Summary of Exceptions to Total Latency

There are usually extra bypass latencies incurred when a result is computed on one type of unit and consumed on another or more generally when a fast bypass is not provided by the Itanium processor. The table below describes most special total latencies on the Itanium processor:

Source Instruction Class	Target Instruction Class	Total Latency
IALU (for I slot instructions only)	LD/ST address register	2
ILOG, PNT, XTD	LD/ST address register	2
LD	LD/ST address register	3 (L1 hit)
IALU, ILOG	MMMUL, MMSHF, MMALU	3
LD	MM operation	3 (L1 hit)
MM operations	IALU, ILOG, ISHF, ST, LD	If scheduled <4 cycles apart, 10 clock flush; If scheduled >=4 cycles apart) 4 clocks
TOBR, TOPR, TOAR (pfs only)	BR	0
FRBR, FRCR, FRIP, FRAR (FRxx)	MMMUL, MMSHF, MMALU	FRxx + 1
FMAC	FMISC, FCVTXF, XMA	7
	FMAC[a]	7
SFMAC	SFMISC	7
	SFMAC[a]	7
SFxxx (32-bit parallel FPt)	Fxxx (64/82-bit FP)	SFxxx + 2 cycles
Fxxx (64/82-bit FP)	SFxxx (32-bit parallel FP)	Fxxx + 2 cycles
All FP register writing instructions except loads	STF, FRFR	8

a. In general, the FMAC (SFMAC) latency is 5 cycles when being consumed by another FMAC. However, when the result of the producer FMAC (either half, in the case of SFMAC) is a NaN, infinite, zero, or NaTVal, the latency to the consumer FMAC is 7 cycles. The algorithm used to check for these conditions is approximate and thus there are other rare cases where the 7-cycle latency may occur even when one of these conditions does not actually occur.

4.5 Predicates and Bypassing

In the presence of predication, the actual dependences between non-unit latency instructions are sometimes determined at execution time rather than at compile time. *Predication does not affect bypass latency of unit latency instructions*. The discussion below provides a high-level of detail about the effect of predication on bypassing, but the overall effect on performance due to these cases is generally very small.

Consider the code below. If `p1` were false at runtime, there would be no dependence between instructions B and C, so C could issue in cycle 2. On the other hand, if `p1` were true, then instruction C would be dependent upon B and would have to stall until B completed:

```
      cmp.eq  p1,p2=r5,r4 ;;  // Cycle 0: instr A
(p1)  ld8     r1=[r3] ;;      // Cycle 1: instr B
      add     r3=r1,r2        // Cycle ?: instr C
```

Whether the predicate is true or false, if the distance between the producer of the predicate and the consumer of a bypassed general register is less than two cycles, the consuming instruction will stall until the second cycle after the compare. In the code below, the consumer (`add`) is predicated while the producer (`ld8`) is not. The compare and consumer are only scheduled one cycle apart. The `add` will stall until cycle 2 while waiting to see if the predicate is true:

```
      cmp.eq  p1,p2=r5,r4 // Cycle 0
      ld8     r1=[r3] ;;  // Cycle 0
(p1)  add     r3=r1,r2    // Stalls until cycle 2
                          //  even if p1 is false
```

A special case where there must be three cycles between the compare and the consumer of a bypassed register value is shown below:

```
      cmp.eq  p1,p2=r5,r4 ;;// Cycle 0
(p1)  add     r1=r2,r3 ;; // Cycle 1
      ld8     r6=[r1]     // Cycle 3
```

During execution of this code, the `ld8` will stall one cycle. To avoid the stall, one additional cycle must be scheduled somewhere between the cmp and `ld8`. This special case applies when these three conditions are met:

1. A predicated address computation is feeding a load, and
2. The address computation was performed in an M slot, and
3. The address computation was an IALU (but not ILOG or PNT) instruction.

5.0 Memory Hierarchy

The Itanium processor memory hierarchy includes:

- The first level data cache (L1-D)
- The first level instruction cache (L1-I)
- The second level unified cache (L2)
- The third level unified cache (L3)

- The first-level data translation lookaside buffer (L1-DTLB)
- The second-level data translation lookaside buffer (L2-DTLB)
- The instruction translation cache (ITLB)
- The main memory (frontside) bus

5.1 L1 Data Cache

The L1 data cache is 16 Kbytes, 4-way set associative, write through, no write allocate with 32-byte lines. The L1 cache can sustain 2 loads, 2 stores, or 1 load and 1 store per clock. Integer loads that hit in L1 have a 2 cycle latency to most consumer operations. Floating-point loads always bypass the L1 data cache.

Stores that write values that are loaded soon afterwards, referred to as store to load forwarding, may require extra cycles. Any load from an address to which a store was made within the last 3 cycles (inside any part of an aligned 64-bit region) will cause the load to bypass L1 and read from L2.

5.2 L1 Instruction Cache

The L1 instruction cache is 16 Kbytes, 4-way set associative with 32-byte lines.

5.3 L2 Unified Cache

The L2 cache is unified, 96 Kbyte,6-way set associative, write back, and write allocate with 64-byte lines. The L2 has 2 general purpose ports and can sustain up to 2 memory operations per clock or 1 line-fill operation. Integer loads that hit in L2 have a 6 cycle latency to most consumer operations, and FP loads have a 9 cycle latency to consumer operations assuming no L2 contention.

The remainder of the information in this section is highly detailed and only of interest to those working on memory intensive streaming applications.

There are number of *L2 cache pipeline flush* conditions that will affect memory intensive codes. However, for many programs, it is not necessary to optimize for these cases. An L2 pipeline flush means that a memory operation needs to be partially re-executed in the memory pipeline. An L2 pipeline flush adds 6 clocks to the flushed memory reference’s latency, but does not directly affect the main processor pipeline. If L2 cache pipeline flushes occur repeatedly, the main pipeline will eventually stall if too many memory operations back up or operations dependent upon the flushed operations cannot execute.

When a memory reference misses L2 and there are no prior outstanding misses to the same cache line, it is called a *primary* miss. A *secondary* miss is an L2 miss to a line for which a primary miss is already outstanding. L3 can handle one primary L2 miss per clock. Additional primary misses within a clock will cause the second miss to take an L2 pipe flush.

The L2 can tolerate misses on as many as 8 outstanding cache lines at once. Each of the 4 16-byte portions of any one L2 line can have requests from up to 2 memory operations, yielding a maximum of 8 outstanding misses to any one line. Any instruction that would cause either of these maximums to be exceeded will be flushed in the L2 pipeline. Unless there is a pending use for an

L2 request that was flushed, or there is a long dense sequence of such requests, the L2 pipe flush will not immediately affect the main pipeline; however, it does consume L2 cache bandwidth.

Streaming stores can have special performance effects when they occur close together in the execution stream. A store that begins execution and is followed closely by another store may incur L2 stalls or flushes as shown in the table below if they access the same 8-byte aligned region (stores to different regions do not have special behavior):

Size of Subsequent Store	Results When Two Stores Write to the Same 8-Byte Aligned Region Within 3 Clocks of Each Other			
	0 cycles	1 cycle	2 cycles	3 or more cycles
st1, st2	L2 pipeflush	L2 pipeflush	L2 pipeflush	none
st4	1 clk stall	none	L2 pipeflush	none
st8	L2 pipeflush	L2 pipeflush	L2 pipeflush	none

Similarly, if a load follows a store operation within 3 cycles and the load accesses all or part of the same 64-bit aligned region, the load operation will take an L2 pipeflush (thus increasing the latency of the load by the cost of an L2 pipeflush).

5.4 Off Chip, On Package L3

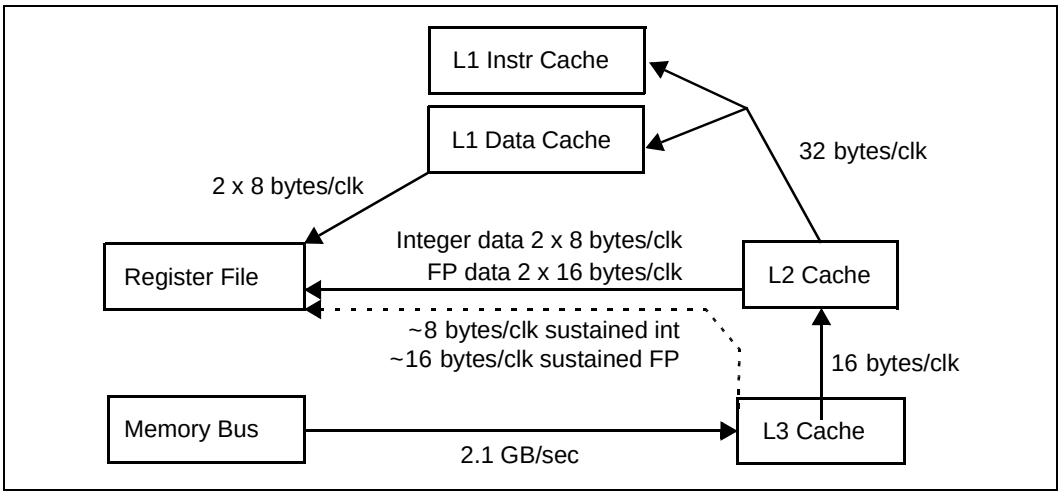
L3 cache hits have a 21 clock latency to integer consumer operations and 24 cycle latency to floating point consumer operations. The size and organization of the L3 vary depending on the particular Itanium package. The maximum bandwidth from L3 cache to L2 is 16 bytes times the core frequency.

5.5 Main Memory Bus

The Itanium processor frontside bus has an approximate maximum bandwidth of 2.1 GB/second.

5.6 Load Bandwidth Summary

The picture below shows approximate peak bandwidth between various sources and destinations of data. The lines in the picture below do not necessarily represent actual busses or widths of busses in the Itanium processor:



The register file can handle up to 2 integer register fills or 2 floating point register fills per clock from the caches.

5.7 Translation Lookaside Buffers

The Itanium processor has two levels of data TLB (L1-DTLB and L2-DTLB), both of which are are fully associative. Misses in the L1-DTLB will cause a 10 cycle main pipe flush penalty. Misses in the L2-DTLB can cost as little as 23 cycles if all needed information is in the L2 data cache.

The L1-DTLB has 32 entries. The L2-DTLB has 96 entries. Page sizes supported by both are: 4k, 8k, 16k, 64k, 256k, 1M, 4M, 16M, 64M, and 256M. Purges supported include all page sizes and 4G. A specification of a purge page size larger than 4G causes all pages in the region to be purged (information on purge behavior is provided for information only — code that uses Itanium-specific behavior to implement required purges will ***not*** run correctly on future processors). Please see the *Intel® Itanium Architecture Software Developer's Manual* for full information on the virtual memory system.

The single-level instruction TLB is fully associative and has 64 entries. For both the instruction and data TLBs, the number of translation registers can be queried by calling PAL.

5.8 Data Speculation, Advanced Loads, and the ALAT

The family of instructions composed of `ld.a`, `ld.c`, and `chk.a` provide the capability to dynamically disambiguate memory addresses between loads and stores. Architecturally, the `ld.c` and `chk.a` instructions have a 0 cycle latency to consuming instructions. However, if a `ld.c` or `chk.a` misses in the ALAT, additional latency is incurred.

A `ld.c` that misses in the ALAT causes a 10-cycle pipeline flush. Additional time is required if the instruction misses the cache during re-execution of the load. A `chk.a` that misses in the ALAT executes a branch to recovery code. On the Itanium processor, this branch is implemented as a trap to the operating system which branches to the recovery code. Thus, the cost of a `chk.a` that misses in the ALAT is likely to be the cost of two branch mispredicts, plus the cost of the recovery code, plus the cost of several control register reads, plus the cost of the return — approximately 50+ cycles plus the cost of recovery code.

On the Itanium processor, the ALAT has 32 entries and is two-way set associative based on the physical target registers of advanced loads. In addition to the physical register number, ALAT entries are tagged with wrap bits that distinguish different instances of the same physical register due to RSE activity.

The virtual to physical register mapping is affected in complex ways by stacked and rotating registers, so the specific physical register that maps to the virtual base register, r32, cannot always be known by a compiler. Therefore, the compiler cannot precisely model the association between rotating and stacked registers in the ALAT when generating code.

Since the Itanium processor's ALAT is 2-way associative, sets of three or more advanced loads may map to the same set of two entries. In the absence of rotating registers, some conflicts can be easy to predict:

```
ld8.a r32=[ r1 ]
ld8.a r48=[ r2 ]
ld8.a r64=[ r3 ]
```

In the example above, the third load will conflict with one of the entries allocated by the previous two advanced loads given the size and structure of the Itanium processor's ALAT. When choosing registers, try to avoid conflicts in the ALAT by recognizing obvious cases like the one above.

Although *register numbers* are used to index and check advanced loads, store *addresses* are used to invalidate potentially overlapping entries. On the Itanium processor, only the 20 low-order bits of a load address are saved in the ALAT to compare against potentially conflicting stores. This partial matching means that stores that do not physically conflict could still be aliased to entries in the ALAT and invalidate them if their low 20 bits match. In situations where a given store is close to a subsequent chk.a or ld.c, fewer than 20 bits are used to determine if a memory conflict occurred:

Conflict Detection Algorithm	Description of Conditions
Always reports a conflict	A store followed by a chk.a in the same cycle
	A ld.a and a ld.c in the same clock
Use low-order 12 bits to determine conflict	A store followed by a ld.c in the same or next cycle
	A store followed by a chk.a in the next cycle
	A snoop followed by a chk.a or ld.c in the next cycle
Use low-order 20 bits to determine conflicts	Other situations

The combined affects of limited associativity and partial address matching mean that if a large number of stores and/or advanced loads occur, entries in the ALAT may be invalidated without ever having an exact memory or register conflict. These characteristics imply that one must carefully consider the number of dynamic stores past which a load can usefully be advanced.

Finally, the use of unaligned advanced loads may cause more ALAT entries than strictly necessary to be purged or for entries not be allocated in the first place. Thus, the use of the ALAT with non-naturally aligned loads is not recommended on the Itanium processor.

5.9 Control Speculation

Almost all Itanium instructions are speculative by default. Loads have speculative and non-speculative versions. When a compiler uses control speculation in the Itanium architecture, it is responsible for inserting `chk.s` instructions to check if recovery code needs to be executed.

When a `chk.s` instruction detects a set NaT bit (deferred exception fault), it branches to recovery code. The total cost of recovering from a NaT fault is the same as for a `chk.a` — approximately 50+ cycles plus the cost of the recovery code.

Note that such deferred exception faults due to control speculation should be very rare and only associated with events that already have long latencies (TLB misses, page faults, etc.) and are thus unlikely to significantly affect performance.

On the Itanium processor, all memory operations that miss in the L1-DTLB will incur the 10 cycle L1-DTLB miss penalty. All speculative and non-speculative versions of loads and stores have this behavior (independently of the NaT deferral policy for the operating system).

6.0 Branching and Control Flow

Itanium instructions help manage branch prediction and control flow. This section describes resources, timing, and restrictions related to branches, branch hints, and prefetching. Section provides concise summaries of features, operation, interpretation, and examples of various hint combinations.

6.1 Branch Prediction

The Itanium processor contains several different structures to predict the directions and target addresses of branches. The particular structure used depends upon the type of branch being predicted, the state of the branch predictor, and the hint completers specified on the branch instruction itself.

A summary of this material can be found in Section 6.4.

The material in this section is highly detailed and is only recommended for those compilers that have already completed advanced classical optimizations, predication, speculation heuristics, and software pipelining. This information is provided for those performing aggressive Itanium-specific optimization related to branch prediction.

6.1.1 Branch Direction Prediction

The Itanium processor has two primary resources to predict whether branches will be taken or not:

- The large branch prediction table (BPT) is used for branches in MFB, MMB, and MIB bundles.
- The smaller multiway branch prediction table (MBPT) is used for branches in MBB and BBB bundles. The MBPT has 1/8 as many multiway branch entries as the BPT has single branch entries.

The use of MBB or BBB bundles results in entries being allocated in the multiway branch predictor regardless of whether the B slots contain hints, nops, or branches. Accordingly, when possible, it is better to use other bundle types that do not require inserting B-slot nops.

Both BPT and MBPT tables are 4-way set associative and are accessed like caches using the bundle address. If a branch misses in these structures, the static prediction encoded in the branch will be used unless the branch hits in the TAC (target address cache) or TAR (target address register) in which case it is predicted taken.

The prediction algorithm used in both MBPT and BPT is a local 2-level predictor with 4 bits of history. The MBPT resources parallel those in the BPT, but each entry in the MBPT has resources for 3 branch slots instead of one. In addition, the BPT and MBPT have fields indicating the type of a branch so that appropriate actions are taken for returns and calls. Allocation in the BPT and MBPT depends on the hints associated with the branches and on the outcomes of the branches, as described in later sections on branch hints and branch predict instructions.

6.1.2 Branch Target Address Prediction

The TAR (target address register) is a fast, 4-entry, fully-associative buffer that is exclusively written to by `brp` instructions with the `.imp` (*important*) completer. A hit in the TAR will cause a branch to be predicted *taken* regardless of whether the branch is in the BPT or MBPT. It also provides the target address for the branch being predicted. The loop predictor can override a TAR prediction if the loop count registers, LC and EC, indicate that a loop will exit.

The TAC (target address cache) is a larger, 64-entry structure that can be written to by either `brp` instructions, branches, or the prediction hardware. If there is a BPT or MBPT hit, the TAC is responsible for providing the target address and the BPT or MBPT will decide whether the branch should be taken or not. The TAC can only hold one address per bundle, so it contains a field to indicate to which slot the target address corresponds. In addition to holding target addresses, a hit in the TAC will return a predicted *taken* result if a branch has already missed in the BPT or MBPT.

The RSB is an address stack on which return addresses are pushed during calls and popped to provide the target addresses for returns.

The BAC computes the correct target address for a branch when an address can be computed quickly, such as the target address of an IP-relative branch. The BAC has two stages called BAC1 and BAC2

6.1.3 Timing Considerations

The mechanism ultimately used to predict a branch's target or direction will affect the number of bubbles inserted into the execution pipeline.

There is no taken-branch bubble associated with a taken branch that hits in the TAR. This is the fastest form of branching possible on the Itanium processor. In general, branches that do not hit in the TAR but are correctly predicted will take one or more pipeline bubbles. However, such bubbles might be absorbed in a decoupling buffer in the Itanium processor pipeline and not have a large effect on performance. Bubbles are more likely to be absorbed if the code coming ahead of the branch has other types of stalls.

There is a one-cycle branch bubble associated with a taken branch whose target was provided by the TAC.

If a branch misses in both the TAR and the TAC and the branch is IP-relative, a branch bubble is inserted and the BAC calculates the correct address. If the first predicted *taken* branch is in the third slot of its bundle, its target address is computed by BAC1, and the branch has a two-cycle bubble. Otherwise, it is computed in BAC2, and the branch has a three-cycle bubble. If the target address computation generates a carry past the twentieth bit, BAC2 will be used.

If a `br.ret` instruction is predicted *taken* by the BPT or MBPT, there will be a one-cycle taken-branch bubble. If the branch misses in the BPT and MBPT and the static prediction encoded in the branch is used, then a two-cycle bubble will be incurred.

Indirect branches that miss in the TAR and TAC have their addresses provided from the top of stack of the RSB, without changing the RSB state. This prediction has a three-cycle branch bubble, and the predicted target address is very likely to be incorrect unless it is a return.

6.2 Affecting Branch Prediction Using Hints

The Itanium architecture has three ways of communicating with the Itanium processor branch-prediction hardware. These are explicit branch predict instructions, move to branch register instructions, and the hints encoded in branch instructions. While syntax of hints is architectural, their effects and timings are microarchitecture specific.

For hints from the `brp` or move to branch register instructions to be effective, they must be scheduled a minimum distance prior to their associated branches. In some cases, this distance is specified in terms of *cycles*. In other cases, it is specified in terms of instruction *fetches*. A cycle is a unit of time (like instruction latency). A fetch is a unit of distance in bundles (statically or dynamically), not time. On the Itanium processor, one fetch is equal to two bundles, which equal one instruction cache line.

6.2.1 Hints Encoded in the Branch Instruction

The sptk, spnt, dptk, and dpnt branch instruction completers indicate what prediction should be used when a branch misses in the dynamic prediction hardware. The differences between the static (sp) and dynamic (dp) hints are the effects they have on allocation in the prediction structures and on prediction.

The static hints (spnt/sptk) indicate not to allocate any space in the BPT. This causes the static prediction specified to be used unless there is an accidental match in the (M)BPT. The following special cases apply:

- The branch will still have an entry allocated in the TAC if it is hinted sptk but is not taken and is not a return.
- If the branch is a call or return and it is hinted sptk, the branch will be allocated in the BPT or MBPT although it will always be predicted *taken*.

The dynamic hints (dpnt/dptk) tell the Itanium processor to use the dynamic prediction hardware, and:

- If a non-return branch is taken, a TAC entry will be allocated.
- The whether-hint is used to predict the branch until the first time the branch is mispredicted. At that point, a BPT or MBPT entry is allocated and used for future predictions.
- If a return or call instruction is taken, a BPT or MBPT entry will be allocated.

The dealloc (clr) hint tells the Itanium processor that regardless of the outcome of a branch, no prediction structure should be allocated or updated. The static prediction encoded in the branch is used, but calls and returns still update the RSB. Note that the clr completer does not actually remove an entry that already exists in the BPT, MBPT, or TAC, it only prevents allocation.

6.2.2 Branch Predict Instructions

Branch-hint instructions can specify branch targets and branch direction. These hints are provided by the branch predict (`brp`) instruction. Since brp instructions are hints only, they have no effect on the correctness of a program and can be ignored by an Itanium architecture implementation. On

the Itanium processor, only those `brp` instructions that are placed in the third slot of a bundle are recognized, otherwise, they are treated as nops.

There is no difference between the loop, sptk or dptk (the exit version is unimplemented) hint types for branch predict instructions on the Itanium processor. These hints cause a TAC entry to be allocated for a branch, thus causing the branch to be predicted taken if there is no match in the (M)BPT. The `brp` instruction must be at least four fetches ahead of the branch it is hinting in order for the TAC write to take effect in time for the branch to use the result.

If the `brp` instruction uses the imp completer, a TAR entry will be allocated in addition to the TAC entry. Since the TAR is read one cycle earlier than the TAC, the `brp.imp` instruction has to come five fetches before its associated branch to be effective.

Although the `brp` instruction has forms for both returns and indirect branches, both forms are unimplemented in the Itanium processor and are thus treated as nops. The functionality for hinting indirect branches is provided by the move to branch instruction.

Since the `brp` instruction only provides the address of the bundle being hinted, it is not possible for software to indicate which slot is being hinted in the case of MBB and BBB (multiway) branches. The Itanium processor assumes that all `brp` instructions refer to the third slot. When a `brp` causes a TAC write, the TAC entry field indicating the corresponding slot is always set to three.

6.2.3 Move to Branch Register Instructions

If a move to branch register instruction has the sptk or dptk completer, it will cause a TAC update.

The Itanium processor pipeline organization demands that the minimum number of instruction cache lines that move to branch register instructions must precede hinted branches is fairly large and depends upon the characteristics of the code being executed and the state of the execution pipeline. Typically, a minimum of 9 cycles must elapse at execution time between a *move to branch* register instruction and a branch for the hints to be seen by the branch (note that the branch instruction will NOT stall waiting for the mov to br — this latency only applies to whether the hints provided by the *mov to branch* will be seen by a consuming branch). More accurately, it might require 9-13 *fetches* of separation to get any benefit.

6.2.4 Last Iteration Prediction: br.ctop and br.cloop

The Itanium processor has a special branch predictor for br.ctop and br.cloop branches to avoid the not taken branch at the end of a loop. This section lists limitations and special case code required for the predictor to be effective.

For both br.cloop and br.ctop, a branch predict instruction (brp.loop.imp) must be used to indicate the taken address of the loop to cause the perfect predictor to be used.

Additionally, for br.cloop (but not br.ctop), the AR[EC] register must contain the value 1 in order for the predictor to correctly predict the last iteration even though there is no architectural connection between br.cloop and the value in the EC register. Thus, it is necessary for code that uses br.cloop and that wishes to use the loop branch exit predictor to initialize the value of EC to 1 prior to entering the br.cloop in addition to using the previously mentioned brp.loop.imp.

The counted loop branch predictor will not avoid the final branch mispredication on extremely short loops while the first few iterations work their way through the pipeline.

6.3 Affecting Instruction Prefetching Using Hints

Instruction prefetching can help reduce the latency of an L1-I cache miss and in some cases eliminate it. On the Itanium processor, prefetches are requested using:

- Hints encoded in branch instructions

While up to two `brp`s can be issued in each cycle, at most one prefetch hint can be processed per clock. On average, trying to issue more than one prefetch each cycle will ultimately cause some of the prefetches to be dropped. Once a prefetch is issued and is queued, it will eventually be sent to the cache unless it is canceled or flushed due to a mispredicted branch. If the prefetch queue is full, then later prefetches will be dropped.

On the Itanium processor, only those `brp` instructions that are placed in the third slot of a bundle are recognized, otherwise, they are treated as nops.

Some Itanium instructions include prefetch hints to specify that *many* or *few* instruction cache lines should be fetched. The definitions of *many* and *few* are implementation dependent. For the Itanium processor, two prefetching algorithms are implemented: *streaming* prefetch and *line* prefetch. Line prefetches retrieve two L1 cache lines, while streaming prefetches continuously fetch lines until a predicted taken branch is encountered by the main pipeline.

6.3.1 Streaming Prefetch Hints on Branch Instructions

Prefetch hints on the Itanium processor that are specified on branch instructions only cause a prefetch if the branch is predicted taken. The set of instructions that are prefetched begins at the first L2 cache line *after* the target of the branch.Prefetching continues until the next predicted taken branch is encountered. However, the streaming prefetcher is decoupled from the main pipeline and continuously fetches until the main pipeline detects a predicted taken branch. That means the prefetcher can fetch significantly beyond the end of the next region if there are many stalls in the code being executed.

Only a branch in the third slot of a bundle can start a streaming prefetch.

6.4 Summary of Branch Prediction and Prefetching

Since there is a large and complex amount of information regarding branch prediction, instruction prefetching, and hinting, this section provides the information in bulleted lists. More complete descriptions have been provided already in section.

6.4.1 Syntax Summary and Interpretation

This section summarizes Itanium instruction syntax (and its interpretation on the Itanium processor) for specifying hints on branches, branch predict instructions, and move to branch instructions.

Interpretation of Hints on Branch Instructions

This subsection describes how the Itanium processor hardware interprets various hint completers on branch instructions and provides several examples.

Whether hints:

- *sp*XX – Don’t allocate space in (M)BPT for this branch
- *dp*XX – Allocate space in (M)BPT after the first misprediction of this branch
- XX*nt* – Predict this branch not taken if no (M)BPT entry found
- XX*tk* – Predict this branch taken if no (M)BPT entry found and write target into the TAC

Deallocation hints:

- *clr* – Never allocate space in the BPT, TAC, or TAR when this branch is executed. Entries could still be present from aliased entries or brp/move to branch register instructions.
- No completer – Allocate space according to other rules

Prefetch hints:

- No completer or *few* - No prefetching
- *many* – Start a streaming prefetch at the second level cache line after the branch target

Other rules:

- A TAC entry is allocated the first time a branch is taken unless *clr* or *spnt* is specified or the branch is `br.ret`
- For *sptk*-encoded br.calls and `br.ret`, (M)BPT will be allocated, but the branch will always be predicted *taken*

Example: `br.cond.sptk L1`

- If no (M)BPT entry, predict it *taken*, else use (M)BPT prediction
- Execution of this instruction will not cause a new (M)BPT entry allocation
- It is possible that a `brp` or alias with another branch could cause a (M)BPT hit
- If the branch is taken, a TAC entry is allocated
- No prefetching occurs

Example: `br.cond.dpnt.many L1`

- If no (M)BPT entry is found for this branch, predict *not taken*, otherwise use (M)BPT prediction
- If this branch mispredicts (is taken), allocate a (M)BPT entry
- If the branch is taken, a TAC entry is allocated
- If the branch is predicted taken, start a streaming prefetch at the second level cache line after the branch

Example: `br.cond.sptk.few.clr L1`

- If no (M)BPT entry is found for this branch, predict it taken, otherwise use (M)BPT prediction
- The execution of this instruction will not cause a (M)BPT entry to be allocated nor will any existing entries be updated
- It is possible that a `brp` or alias with another branch could cause a (M)BPT hit
- No prefetching is performed
- No TAC entry is allocated

Interpretation of BRP Instructions

This section describes the syntax of brp instructions and how specific hint completers are interpreted by the Itanium processor hardware. Several examples are given at the end of the section.

IP-relative whether hints:

- *sptk*, *dptk*, *loop* – TAC entry allocated, no (M)BPT update
- *exit* – No effect on (M)BPT or TAC

Hints for indirect branches:

- Unimplemented on the Itanium processor
- Hints on move to branch instructions support this functionality

Importance hints:

- *imp* – Allocate a TAR entry in addition to TAC
- No completer – Allocate space according to other rules

Example: `brp.loop.imp target25, L1`

- Allocate a TAR entry for the branch at label L1 that points to target25

Example: `brp.dptk target25, L1`

- Allocate a TAC entry for the branch at label L1 that points to target25

Interpretation of Hint on Move to Branch Register Instructions

This section describes the syntax and interpretation of hints provided as part of the move to branch instruction.

Whether hints:

- *sptk*, *dptk* – Allocate TAC entry, no (M)BPT change
- No completer – No TAC allocation

Importance hints:

- *imp* – Allocate a TAR entry in addition to TAC
- No completer – Allocate space according to other rules

6.4.2 Branch Operation, Prediction, Timing and Resources

This section summarizes the resources and rules that control branch prediction for direction and target determination on the Itanium processor. These rules completely determine how branching is functionally performed on the Itanium processor. Beyond the behavior described here, branch hints cannot directly influence the operation of a specific branch. They can only influence the contents and allocation of branch resources that then directly influence branch behavior.

Branch direction prediction resources summary:

- The large branch predict table (BPT) used for predicting MIB, MFB, and MMB bundles

- The smaller multiway branch predict table (MBPT) used for predicting MBB and BBB bundles

Branch direction determination summary:

- If a matching entry is found in the TAR, the branch is predicted *taken* except when EC/LC indicate end of loop in counted loop. In that case, a matching TAR entry will be overridden and predicted *not taken*.
- If an entry is found in the BPT (for MIB, MFB, or MMB bundles) or the MBPT (for MBB/BBB bundles), that prediction is used
- If no (M)BPT entry exists, but there is a hit in the TAR or TAC, predict the branch *taken*
- If there are no matching (M)BPT, TAC, or TAR entries for a branch, use the whether-hint encoded in the branch instruction

Branch target prediction resources summary:

- 64-entry target address cache (TAC) (only one entry per bundle).
- 4-entry target address registers (TAR) (only one entry per two bundles).
- 8-entry return stack buffer (RSB) for return instructions.
- For IP-relative branches, there are two branch address correctors (BAC1 and BAC2) that can compute the correct address after a 2 or 3 cycle bubble for branches that miss in both the TAC and TAR.

Branch target determination summary:

- If the branch is a return, use the address on the top of the RSB
- If there is a hit in the TAR, use that address
- If there is a hit in the TAC, use that address
- If neither TAR or TAC hit, and the branch is IP-relative, use BAC1 or BAC2 to compute the address
- If non-IP-relative branches, mispredict

Branch Timing/Bubble Summary:

- Branch mispredictions cause 9 cycles of bubbles in the pipeline.
- Any branch which hits in the TAR is predicted taken and incurs no bubbles.
- Any branch which hits in the TAC and is predicted taken incurs one bubble.
- For IP relative branches that miss the TAC and TAR, BAC1 computes the correct address and incurs 2 bubbles.
- For IP relative branches that miss in the TAC and TAR, BAC2 computes the correct address when BAC1 is not used and incurs 3 bubbles.
- Return instructions whose address is provided from the RSB incur one bubble if the prediction came from the BPT or MBPT or 2 bubbles if the prediction came from the whether hint encoded on the branch.
- The loop predictor can override the TAR when the LC/EC indicate the loop is finished.

7.0 Latency Information for Uncommon Operations

This section contains information on latencies for less commonly used instructions or whose latency cannot easily be described with a single number (it is dependent upon pipeline, memory, TLB state, etc.). Many of the latencies here are provided for specialized uses that system level software or compilers might use. Since latency conditions for such cases are complex, treat the numbers provided in this section as approximate, given the understanding that latencies may be influenced by the state of the processor at the time the instructions are executed.

From AR Register Latency	General Description	Latency (cycles)
FRAR_M, FRAR_I	mov =ar.ccv	6
	mov =ar.unat	6
	mov =ar.rnat	6
	mov =ar.kr[0-7]	13
	mov =ar.bsp	13
	mov =ar.bspstore	13
	mov =ar.rsc	13
	mov =ar.fpsr	13
	mov =ar.eflag	13
	mov =ar.csd	13
	mov =ar.ssd	13
	mov =ar.cflg	13
	mov =ar.fsr	13
	mov =ar.lc	2
	mov =ar.ec	2
	mov =ar.pfs	2
	mov =ar.itc	38
	mov =ar.fir	38
	mov =ar.fdr	13
	mov =ar.fcr	38

There are limitations on the number of outstanding `mov ar` instructions that can be outstanding, however, this effect should not be seen in normal code.

Move to AR Latencies	General Description	Latency (cycles)
TOAR_I, TOAR_M	mov ar.ccv=	5
	mov ar.unat=	5
	mov ar.lc=	1
	mov ar.ec=	1
	mov ar.rnat=	9 (to spill/fill) 5 (to explicit read)
	mov ar.kr=	2 (to explicit read)
	mov bsp=	n/a
	mov ar.bspstore=	10 (to spill/fill) 5 (to implicit ar.rnat accesses)
	mov ar.rsc=	10 (to spill/fill) if immediate form: 1 (to implicit ar.bspstore or ar.rnat accesses) if register form: 10 (to implicit ar.bspstore or ar.rnat accesses)
	mov ar.fpsr=	9 (to FP op) 2 (to explicit read)
	mov ar.eflag=	2
	mov ar.csd=	2
	mov ar.ssd=	2
	mov ar.cflg=	2
	mov ar.itc=	35
	mov ar.fir=	4
	mov ar.fcr=	4
	mov ar.fsr=	23
	mov ar.pfs=	0 (to br.ret)
	mov ar.fdr=	2

There are limitations on the number of outstanding `mov ar` instructions that can be outstanding, however, this effect should not be seen in normal code.

There are limitations on the number of outstanding `mov cr` instructions that can be outstanding, however, this effect should not be seen in normal code.

Semaphore Latency	General Description	Latency (cycles)
SEM	cmpxchg	Approximately the latency of L2, L3, or memory + 5 clocks. These operations are not pipelined.
	xchg	
	fetchadd	

On the Itanium processor, semaphore operations stall the pipeline (latency cannot be hidden).

Various System Instruction Latencies	General Description	Latency (cycles)
SYST_I, SYST_I0, SYST_B, SYST_B2, SYST_M0, SYST_M	alloc	1 to reg argument. Stalls depends on RSE state.
	flushrs	Stall depends on RSE state
	loadrs	Stall depends on RSE state
	probe, probe.fault	Variable
	ttag	13
	thash	13
	tpa	6
	tak	6
	fc	Variable
	mov =pkr	13
	mov =rr	13
	mov =psr	13
	mov =pmc	38
	mov =pmd	38
	mov =ibr	38
	mov =dbr	38
	mov =cpuid	38
	mov pkr=	10 (to srlz)
	mov rr=	10 (to srlz)
	mov pmc=	35 (to srlz)
	mov pmd=	35 (to srlz)
	mov ibr=	35 (to srlz)
	mov dbr=	35 (to srlz)
	rum	4 (to use)
	sum	4 (to use)
	mov psr.um=	4 (to use)
	mov psr.l=	5 (to srlz)
	rsm	5 (to srlz)
	ssm	5 (to srlz)
	itc.i, itc.d	Variable
	itr.i, itr.d	
	ptr.i, ptr.d	
	ptc.l	
	ptc.e	
	ptc.g, ptc.ga	
	invala.e	
	invala	

There are limitations on the number of outstanding system instructions that can be outstanding, however, this effect should not be seen in normal code.

Move to CR Latencies	General Description	Latency (cycles)
TOCR	mov cr.isr=	5 (to srlz)
	mov cr.iip=	5 (to srlz)
	mov cr.iipa=	5 (to srlz)
	mov cr.iim=	10 (to srlz)
	mov cr.iva=	5 (to srlz)
	mov cr.itir=	5 (to itc/itr) 10 (to srlz)
	mov cr.ifa=	5 (to itc/itr) 10 (to srlz)
	mov cr.ifs=	10 (to srlz)
	mov cr.ipsr=	10 (to srlz)
	mov cr.dcr=	5 (to srlz)
	mov cr.iha=	10 (to srlz)
	mov cr.pta=	10 (to srlz)
	mov cr.itm=	35 (to srlz)
	mov cr.lid=	35 (to srlz)
	mov cr.tpr=	35 (to srlz)
	mov cr.eoi=	35 (to srlz)
	mov cr.itv=	35 (to srlz)
	mov cr.pmv=	35 (to srlz)
	mov cr.cmcv=	35 (to srlz)
	mov cr.lrr[0-1]=	35 (to srlz)

There are limitations on the number of outstanding `mov cr` instructions that can be outstanding, however, this effect should not be seen in normal code.

From CR Register Latency	General Description	Latency (cycles)
FRCR	mov =cr.isr	2
	mov =cr.iip	2
	mov =cr.iipa	2
	mov =cr.iim	2
	mov =cr.iva	2
	mov =cr.itir	13
	mov =cr.ifa	13
	mov =cr.ifs	13
	mov =cr.ipsr	13
	mov =cr.dcr	13
	mov =cr.iha	13
	mov =cr.pta	13
	mov =cr.itm	38
	mov =cr.lid	38
	mov =cr.ivr	38
	mov =cr.tpr	38
	mov =cr.eoi	38
	mov =cr.irr[0-3]	38
	mov =cr.itv	38
	mov =cr.pmv	38
	mov =cr.cmcv	38
	mov =cr.lrr[0-1]	38