Itanium

Lin Gao

Contents

Chapter 1

Introduction

The Itanium is an IA-64 microprocessor developed jointly by Hewlett-Packard (HP). The Itanium architecture is base on explicitly parallel instruction computing (EPIC), where the compiler would line up instructions for parallel execution. It provides a lot of features to support the instruction level parallelism (ILP), such as speculation and prediction. These features will be discussed in detail in the following chapter.

The main structural flaw with the initial Itanium processor was the high latency of its L3 cache. Together with other short-comings, the Itanium was not a competitive product in the market. Itanium 2 is the successor of the first Itanium processor. Although it delivered the performance promise with leadership benchmark results across a wide range of workloads, recent data from market suggests that it's less popular than other 64-bit architectures, like Intel's EM64T.

However, in September 2005, a slew of major companies announced a new "Itanium Alliance" to promote hardware and software development for the Itanium architecture, including HP, Hitachi, Microsoft, Silicon Graphics, Oracle, Intel and a lot more. It indicates a warm move have taken regarding Intel's Itanium chips.

Chapter 2

Itanium Architecture

Figure 2.1 gives an overview of the Itanium architecture[1, 2].

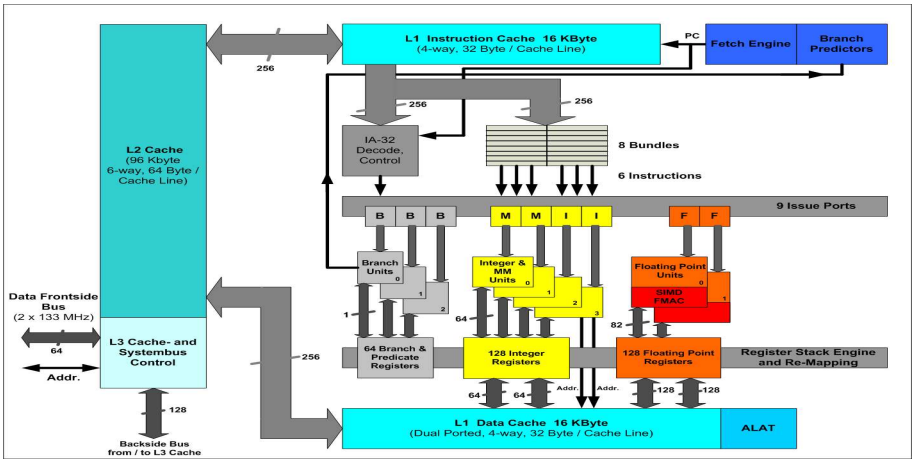

Figure 2.1: The Itanium Architecture

The IA-64 architecture was designed to overcome the performance limitations of traditional architectures and provide maximum headroom for the future. To achieve this, IA-64 has an array of features to extract greater ILP including speculation, predication, large register files, a register stack, advanced branch architecture, and many others. In this chapter, we will briefly introduce some key features in Itanium architecture.

2.1 Register State, Stack and Rotation

Itanium provides a rich set of system register resources for process control, interruptions handling, protection, debugging, and performance monitoring. Only registers belonging to application register state are visible to application programs. Figure 2.2 shows the application register state, which is part of the set of all defined privileged system register resources. The following

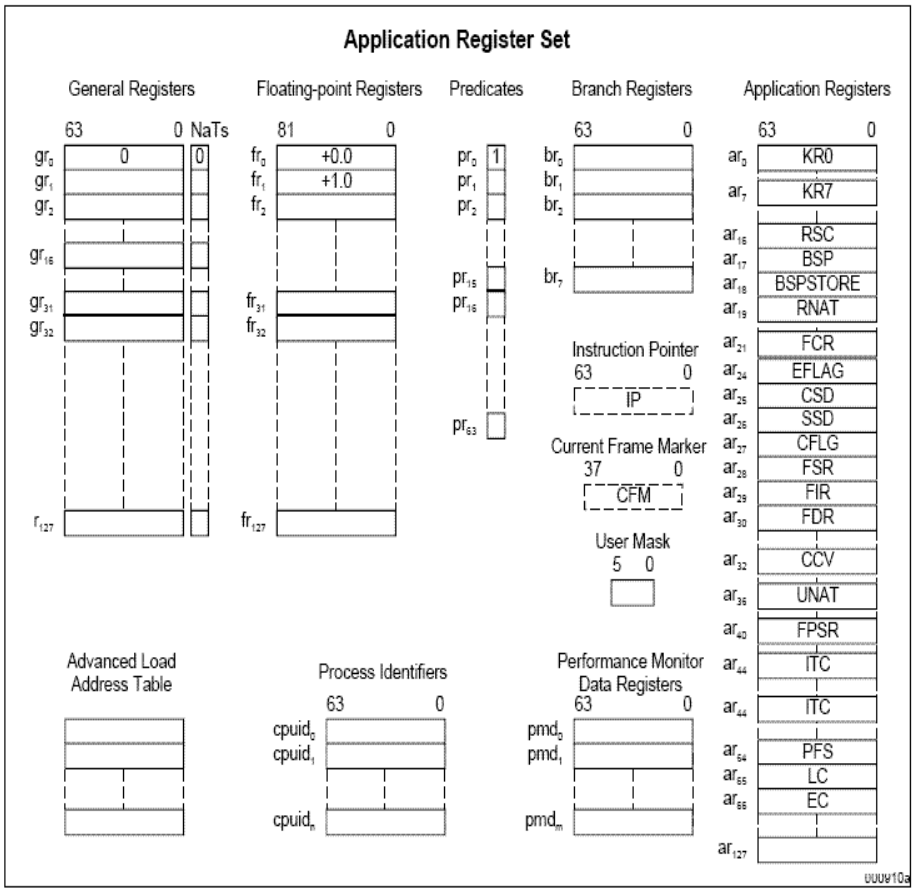

Figure 2.2: Itanium Application Register State

is a list of the registers available to application programs:

- **General Registers (GRs)** A set of 128 general purpose 64-bit registers (GR0-GR127) provide the central resource for all integer and integer multimedia computation. Each general register has 64 bits of

normal data storage plus an additional bit, the NaT bit (Not a Thing), which is used to track deferred speculative exceptions. GR0-GR31 are *static* and visible to all procedure, while GR32-GR127 are *stacked* and is local to each procedure. A programmable sized portion of stacked register can be defined to be *rotate*. Rotating registers are hardware support for software-pipelined loops.

- **Floating-point Registers (FRs)** A set of 128 82-bit floating-point registers (FR0-FR127) are used for all floating-point computation. FR0-FR31 are *static* and FR32-FR127 are *rotating* FRs. Deferred speculative exceptions are recorded with a special register value called NaTVal (Not a Thing Value).

- **Predicate Registers (PRs)** A set of 64 1-bit predicate registers (PR0-PR63) are used to hold the results of IA-64 compare instructions. PR0-PR15 are *static* and used in conditional branching. PR16-PR63 are *rotating*.

- **Branch Registers (BRs)** A set of 8 64-bit branch registers (BR0-BR7) are used to hold branching information, which specify the branch target address for indirect branches.

- **Instruction Pointer (IP)** The IP holds the address of the bundle which contains the current executing IA-64 instruction.

- **Current Frame Marker (CFM)** Each general register stack frame is associated with a frame marker. The frame marker describes the state of the general register stack, such as size of rotating portion of stack frame and rotating register base for GRs. The CFM holds the state of current stack frame.

- **Application Registers (ARs)** The application register file includes special-purpose data registers and control registers for application-visible processor functions for ISA. There are 128 64-bit application registers (AR0-AR127). For example, AR65 is the loop count register (LC), which is decremented by counted-loop-type branches.

- **User Mask (UM)** The user mask control s memory access alignment, byte-ordering and user-configured performance monitors.

As described above, the stacked subset of GRs is local to each procedure and may vary in size from zero to 96 registers beginning at GR32. The register stack mechanism is implemented by renaming register addresses as

a side-effect of procedure calls and returns. The implementation of this rename mechanism is not visible to application programs. Register stack is automatically saved and restored by the Register Stack Engine (RSE) without explicit software intervention. RSE use spare memory bandwith to perform register spill and reload operations in the background when necessary. Register spill and reload may cause RSE traffic. Advanced register allocation[5] may reduce the RSE traffic.

A register stack frame is a set of stacked GRs that are visible to a given procedure. The frame is further partitioned into two variable-size area: the local area and the output area. After a call, the size of the local area of the newly actived frame (related to the callee) is zero and that of the output area is equal to the size of caller's output area and overlays the caller's output area. The *alloc* instruction can be used to dynamically define the local and output area of current frame. Figure 2.3 depicts the behavior of the register stack on a procedure call from procA (caller) to procB (callee).

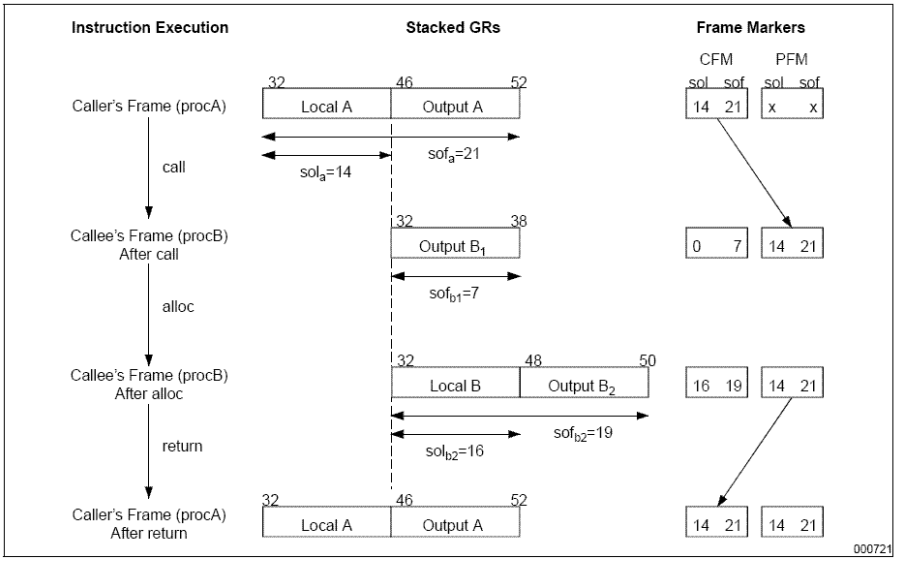

Figure 2.3: Register Stack Behavior on Procedure Call and Return

Register rotation is a feature of the Itanium to support software pipelining. A fixed sized area of the predicate and floating-point register files (PR16-PR63 and FR32-FR127) and a programmable sized area of general register file are defined to "rotate". The general register rotating area starts at GR32 and overlays the local and output area depending on their relevent

Instruction Type	Description	Execution Unit Type
A	Integer ALU	I-unit or M-unit
I	Non-ALU integer	I-unit
M	Memory	M-unit
F	Floating-point	F-unit
B	Branch	B-unit
L+X	Extended	I-unit/B-unit

Table 2.1: Instruction Type and Execution Unit Type

sizes. The size of rotating area in GR file is determined by the *alloc* instruction, the size must be either zero or a multiple of 8. Registers are rotated toward larger register numbers by one register position in a wraparound fashion, and it occurs when a software-pipelined loop type branch is executed. Rotation is implemented by renaming register numbers based on the value of a rotating register base (rrb) contained in CFM. The operation of the rotating register rename mechanism is transparent to software.

2.2 Instruction Decode

Figure 2.1 gives a brief introduce about the instruction decoding. Each IA-64 instruction is categorized into one of six types as showed in Table 2.1. Each instruction type may be executed on one or more execution unit types. Extended instructions are used for long immediate integer (on I-unit) and long branch (on B-unit) instructions.

Three instructions are grouped together into a 128-bit container called *bundle*. Each bundle contains three 41-bit *instruction slots* and a 5-bit template field. Instrutions cannot be arbitrarily grouped into bundles due to the dependences and resource constraints. The template field specifies the mapping of instruction slots to execution unit types. There are 12 basic template types:

MII, MI|I, MLX, MMI, M|MI, MFI, MMF, MIB, MBB, BBB, MMB, MFB

Each letter in a template type represents the mapping of instruction to execution unit. For example, template MFI specifies that instructions in slot 0, slot 1 and slot 2 of current bundle have to be dispatched to M-unit, F-unit and I-unit respectively. A vertical bar in a template denotes an intra-bundle *stop*, which indicates to the hardware that one or more instructions

before the stop may have certain kinds of resource dependencies with one or more instructions after the stop. Each basic template type has two version: one with a stop after the slot 0 and one without. Instructions must be placed in slots corresponding to their instruction types based on the template specification, except for A-type instructions that can go into either I or M slots. Bundle templates enable Itanium processors to dispatch instructions with simple instruction decoding, and stops enable explicit specification of parallelism.

The Itanium architecture is designed to sustain the execution of 6 instructions per cycle (which equals to 2 bundles per cycle), although it has enough issue control and data paths to issue 9 instructions per cycle. Having 9 issue ports (2 I, 2 F, 3 B and 2 M issue ports) allows ports to be dedicated to specific functions and to increase instruction dispersal efficiency.

2.3 Speculation

Memory can only be accessed through load and store instructions and special semaphore instructions in Itanium architecture. programmer-controlled speculation is also supported for hiding memory latency. Two kinds of speculation are control speculation and data speculation.

2.3.1 Control Speculation

Control speculation allows loads and their dependent uses to be safely moved above branches. NaT bits attached to GRs and NaTVal values for FRs enable control speculation. In case where data read by speculative load turns out not to be needed, its results are simply discarded, otherwise memory latencies are overlapped by the execution of other instructions. When a speculative load causes an exception, the exception is deferred by setting the NaT bit on the destination register (or writing NaTVal into the FR) and propagating the setting across its dependent uses until a subsequent non-speculative instruction checks for or raises the deferred exception.

An additional mechanism[6] is defined that allows the OS to control the exception behavior of speculative loads. The OS has the option to select which exceptions are deferred automatically in hardware and which exceptions will be handled (and possibly deferred) by software.

2.3.2 Data Speculation

Data speculation allows loads to be moved above possibly conflicting memory references. A store that cannot be statically disambiguated relative to a particular load is said to be ambiguous relative to that load. In such case, compiler cannot change the original order of load and store specified in the program. Itanium provides a special kind of load instruction called an *advanecd load*, that can be scheduled to execute earlier than one or more stores that are ambiguous relative to that load. Compiler can also speculate operations dependent on the advanced load and later insert a check instruction that will determine whether the speculation is successful or not. The check instruction for the advanced load can be placed anywhere the original non-data speculative load could have been scheduled. The decision to perform data speculation is highly dependent on the possibility and the cost of recovering from an failed data speculation.

An additional structure called *advanced load address table* (ALAT) is used to hold the state necessary for advanced loads and checks. When an advanced load is executed, it allocates an entry in ALAT. Subsequent store to the same memory location will removed the entry from ALAT. Later, when a corresponding check instruction is executed, if a matching entry is found, the data speculation succeeded, otherwise, it failed and the recovery is performed.

2.4 Predication and Branch Hints

Predication is the conditional execution of an instruction based on a qualifying predicate. A qualifying predicate is a predicate register whose value determines whether the processor commits the results computed by an instruction. Predication converts control dependencies to data dependencies by converting branch conditions to predicate registers, this simplifies compiler optimizations and removes associated mispredict penalties.

The execution of most Itanium instructions is gated by a qualifying predicate. Only a few instructions cannot be predicated, such as *alloc* (allocate stack frame), clrrrb (clear rrb) and rfi (return from interruption).

In addition to removing branches through the use of predication, some other mechanism are provided to decrease the branch misprediction rate and the cost of misprediction. An interesting feature of Itanium is its branch hints, which are intended to improve branch prediction by providing information about expected branch behavior to the processor. Branch hints provide the following list of choices:

- statically or dynamically by the hardware
- predict if a branch would be taken or not taken

Chapter 3

Itanium 2

The Itanium 2 processor is the second product in Itanium Processor Family (IPF). There are some changes have been made to Itanium 2 [4, 3]. Figure 3.1 gives the processor block diagram. The difference will be discussed in following sections.

3.1 More Execution Resources

The Itanium architecture is designed to support the parallel execution of 6 instructions per cycle. However, the execution phase of the Itanium processor was usually not able to effectively execute 6 instructions per cycle because of limited execution resources.

Itanium 2 Proc. Execution Units	# Units	Latency
Memory Load Ports	2	1 cycle (L1)
Memory Store Ports	2	NA
ALUs (integer)	6	1 cycle
Integer Units	2	1 cycle
Integer Shift	1	1 cycle
Multimedia ALUs	6	2 cycles
Parallel Multiply Units	1	2 cycles
Parallel Shift-Mask Units	2	2 cycles
FP FMAC (multiply-accumulate)	2	4 cycles
FP FMISC (compares, merge, etc)	2	4 cycles
Branch Unit	3	0-2 cycles

Table 3.1: Itanium 2 Execution Units

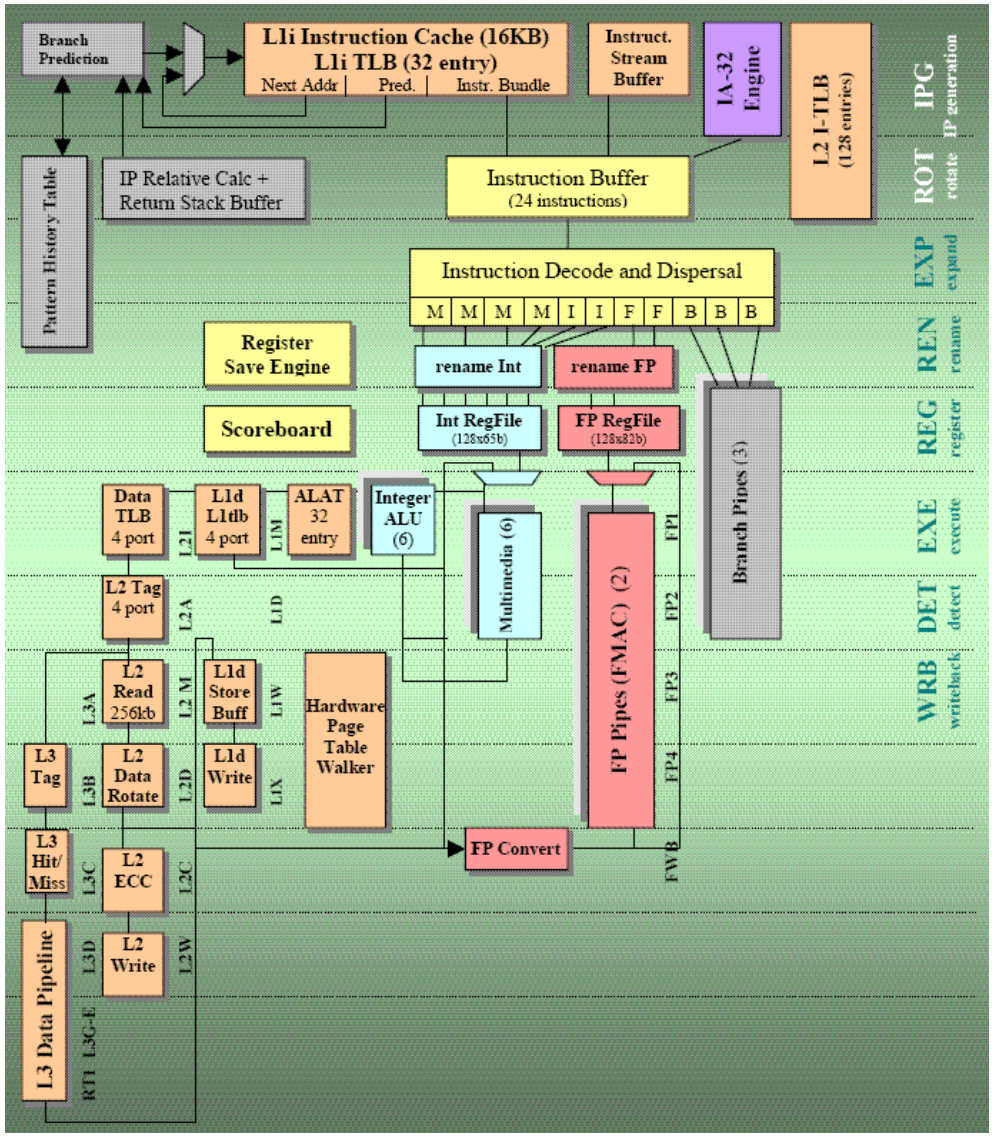

Figure 3.1: Itanium 2 Processor Block Diagram

Table 3.1 lists major execution units in Itanium 2 processor. Compared to Itanium, it provides 2 additional ALU units, 2 additional multimedia units, and 2 additional load/store memory ports. According to the 12 template types, two bundles can include up to 4 memory operations, up to 6 ALU operations, up to 4 integer operations, up to 6 branch operations, and up to 2 FP operations. Providing the additional execution units, Itanium 2 processor can issue nearly all combination of template types in one cycle, and 11 issue ports (compared to 9 issue ports in Itanium) increase the instruction issue efficiency. Figure 3.2 compares the issue efficiency of both architecture. Partially colored boxes indicate that if the first bundle group has only branch hint instructions, then the second instruction bundle may be executed in parallel. If the first bundle contains true branch instructions, the second bundle cannot be executed in parallel.

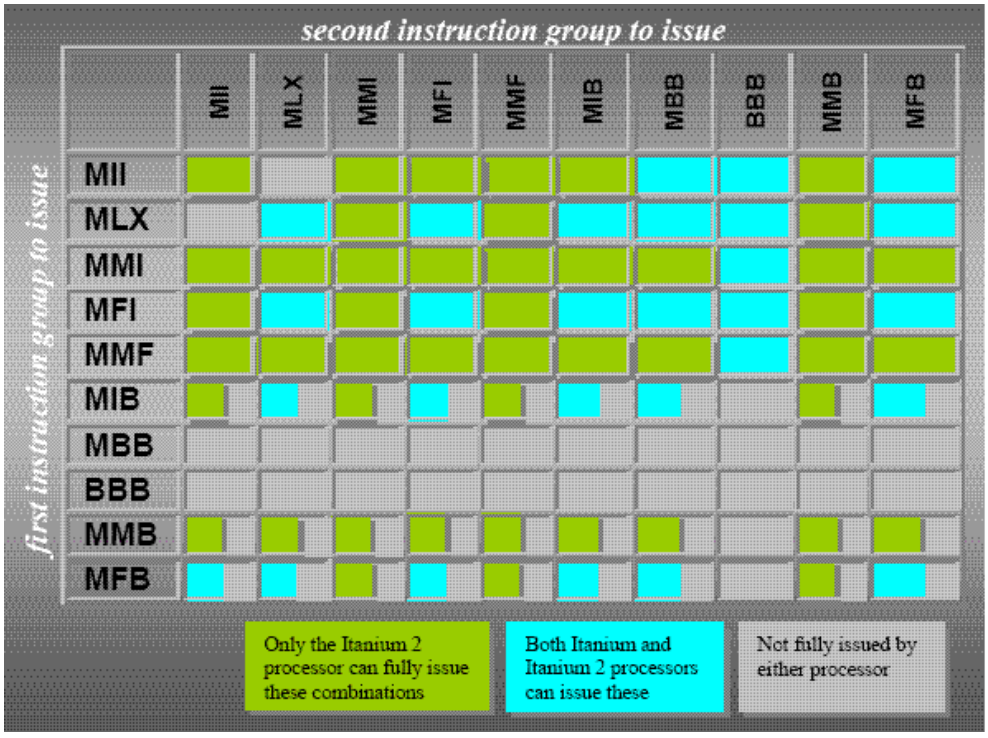

Figure 3.2: Issue Combinations for 2 Bundles

3.2 Cache System Distinction

Itanium 2 have made lots of modification on cache system, including:

- **Cache Latency** Itanium 2 processor cache latencies are nearly half those of the Itanium processor for almost all caches showing in Table 3.2.

caches	Itanium	Itanium 2
L1i	2 cycles	1 cycle
L1d	2 cycles	1 cycle
L2 (I, FP)	6,9 cycles	5,6 cycles
L3 (I, FP)	21,24 cycles	12, 13 cycles

Table 3.2: Cache Latencies Distinction

- **Address** The Itanium 2 processor supports larger virtual addresses (64 bits compared to 50 bits in Itanium) and physical addresses (50 bits compared to 44 bits in Itanium). Expanded addresses are needed by large commercial and technical applications.
- **Line Size** Cache line size in Itanium 2 is doubled for every level of cache. This provides an implicit prefetching that reduces cache misses and effective memory latency.
- **Page Size** Itanium 2 processor supports up to 4GB page sizes, compared to up to 256MB page sizes for Itanium. This allows mapping of large databases and datasets for high-end applications.
- **Bandwidth** Bandwidth from the system bus to the L3, the L3 to the L2, and L2 to the L1s are doubled compared to Itanium. This allows faster cache line transfers.

3.3 Pipeline Enhancements

Figure 3.3 shows the pipeline of Itanium and Itanium 2. The Itanium 2 processor pipeline is 8 stages long, and is two stages shorter than the Itanium processor pipeline. The instruction buffer decouples the $frontend$ instruction fetching from the *backend* execution stages. Shorter pipeline stages leads to 4-6% performance improvement, caused by less branch misprediction penalties (it takes less time to refill a shorter pipeline after a mispredicted branch is encountered).

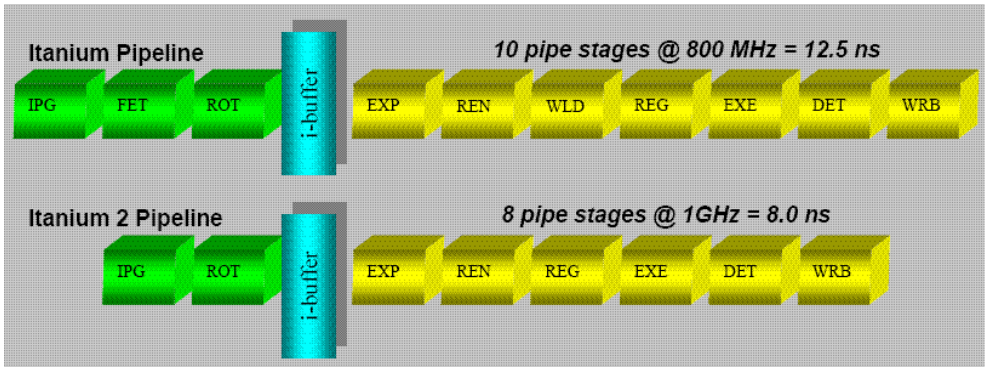

Figure 3.3: Itanium and Itanium 2 pipelines

The reduction in frontend is the result of 1 cycle latency of instruction fetching from L1i, which allows the instructions and control information used to generate the next instruction address to be read out in 1 cycle.

WLD (word-line decode) stage in Itanium was not needed in Itanium 2 because of more aggressive circuit design techniques used in register file design.

Providing all these changes to the Itanium architecture, the benchmark results across a wide range of workloads show that Itanium 2 delivers promising performance.

Bibliography

[1] Intel ia-64 architecture software developer's manual – volume 1: Ia-64 application architecture. In *Document Number:245317-002*, 2000.

[2] Intel ia-64 architecture software developer's manual – volume 2: Ia-64 system architecture. *Document number: 245318-002*, 2000.

[3] Inside the intel itanium 2 processor. In *Hewlett Packard Technical White Paper*, 2002.

[4] Cameron McNairy and Don Soltis. Itanium 2 processor microrchitecture. *IEEE Micro*, 23(2):44–55, 2003.

[5] Alex Settle, Daniel A. Connors, Gerolf Hoflehner, and Dan Lavery. Optimization for the intel itanium architecture register stack. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, pages 115–124. IEEE Computer Society, 2003.

[6] Rumi Zahir, Jonathan Ross, Dale Morris, and Drew Hess. Os and compiler considerations in the design of the ia-64 architecture. In *Architectural Support for Programming Languages and Operating Systems, Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 212–221. ACM Press, 2000.