SCI INSTITUTE
TECHNICAL REPORT

A Survey of the Itanium Architecture from a Programmer's Perspective

Christiaan Paul Gribble and Steven G. Parker

UUSCI-2003-003

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA

August 30, 2003

Abstract:

The Itanium family of processors represents Intel's foray into the world of Explicitly Parallel Instruction Computing and 64-bit system design. This survey contains an introduction to the Itanium architecture and instruction set, as well as some of the available implementations. Taking a programmer's perspective, we have attempted to distill the relevant information from a variety of sources, including the Intel Itanium architecture documentation.

In varying levels of detail, we cover the important characteristics of the Itanium architecture, a large portion of the Itanium instruction set, program performance factors and optimizations, and several of the available Itanium implementations. While this survey does not provide exhaustive discussions of these topics, we hope that it will serve as a practical introduction to creating new applications for the Itanium architecture.

THE UNIVERSITY OF UTAH

A SURVEY OF THE ITANIUM ARCHITECTURE FROM A PROGRAMMER'S PERSPECTIVE

Christiaan Paul Gribble
Steven G. Parker

Scientific Computing and Imaging Institute
University of Utah
August 2003

Copyright © 2003
Christiaan Paul Gribble and Steven G. Parker
All Rights Reserved

Contents

1 The Itanium Architecture **1**

2 Instruction Set Architecture **3**
- 2.1 Information Units and Data Types . . . 3
  - 2.1.1 Integers . . . 3
  - 2.1.2 Floating-Point Numbers . . . 3
  - 2.1.3 Alphanumeric Characters . . . 5
- 2.2 Instruction Formats . . . 5
- 2.3 Instruction Classes . . . 6
- 2.4 Addressing Modes . . . 7
  - 2.4.1 Immediate Addressing . . . 7
  - 2.4.2 Register Direct Addressing . . . 7
  - 2.4.3 Register Indirect Addressing . . . 7
  - 2.4.4 Autoincrement Addressing . . . 7

3 Architectural Registers **9**
- 3.1 Instruction Pointer . . . 9
- 3.2 General-Purpose Registers . . . 9
- 3.3 Floating-Point Registers . . . 10
- 3.4 Branch Registers . . . 11
- 3.5 Predicate Registers . . . 12
- 3.6 Application Registers . . . 12
- 3.7 System Information Registers . . . 12
- 3.8 Other Processor Registers . . . 13

4 Itanium Assembler Statements **14**

5 Integer Instructions **16**
- 5.1 Arithmetic Instructions . . . 16
  - 5.1.1 Addition . . . 16
  - 5.1.2 Subtraction . . . 16
  - 5.1.3 Shift Left and Add . . . 16
  - 5.1.4 Multiplication and Division of 64-bit Integers . . . 17
  - 5.1.5 Multiplication of 16-bit Integers . . . 17
  - 5.1.6 Special-Case Arithmetic Operations . . . 18
- 5.2 Data Access Instructions . . . 19
  - 5.2.1 Load Instructions . . . 19
  - 5.2.2 Store Instructions . . . 20
  - 5.2.3 Move Long Immediate Instruction . . . 20
  - 5.2.4 Accessing Specialized Registers . . . 20
- 5.3 Miscellaneous Integer Instructions . . . 21
  - 5.3.1 Zero-Extend Instruction . . . 21
  - 5.3.2 Sign-Extend Instruction . . . 21
  - 5.3.3 Instructions for Narrow Data Types . . . 21

6 Comparison and Branching Instructions **22**
- 6.1 Comparison Instructions . . . 22
  - 6.1.1 Signed Comparison . . . 22
  - 6.1.2 Unsigned Comparison . . . 22
  - 6.1.3 Unconditional Comparison . . . 23
- 6.2 Branch Instructions . . . 23

7 Logical and Bit-Level Instructions **25**
7.1 Logical Instructions 25
7.2 Bit-Level Instructions 25
7.2.1 Bit-Shift Instructions 25
7.2.2 Shift Right Pair Instruction 26
7.2.3 Extract and Deposit Instructions 26
7.2.4 Single-Bit Test Instruction 27

8 Floating-Point Instructions **28**
8.1 Arithmetic Instructions 29
8.1.1 Addition, Subtraction, and Multiplication 29
8.1.2 Fused Multiply-Add and Multiply-Subtract 29
8.1.3 Reciprocal and Square Root Approximations 29
8.1.4 Maximum and Minimum Instructions 31
8.1.5 Normalization 31
8.2 Data Access Instructions 31
8.2.1 Load Instructions 31
8.2.2 Store Instructions 33
8.3 Miscellaneous Floating-Point Instructions 33
8.3.1 Floating-Point Compare Instruction 33
8.3.2 Logical Instructions 34
8.3.3 Assembler Pseudo-Ops 34
8.3.4 Floating-Point Merge Instruction 34
8.3.5 Floating-Point Value Classification 35
8.4 Floating-Point Operations on Integer Values 36
8.4.1 Data Conversion 36
8.4.2 Integer Multiplication 37

9 Parallel Instructions **38**
9.1 Integer Instructions 38
9.2 Floating-Point Instructions 38

10 Structured Programming Constructs **39**
10.1 If...Then...Else Structures 39
10.1.1 Standard Implementation 39
10.1.2 Predicated Implementation 39
10.1.3 Nested If...Then...Else Structures Using Predication 40
10.2 Case Selection Structures 41
10.3 Loop Structures 41
10.3.1 Counter-controlled Loops 41
10.3.2 Loops Controlled by an Address Limit 42
10.3.3 Loops with a Conditional Entrance 42
10.3.4 Using the Loop Count Register 42

11 Using Procedures and Functions **44**
11.1 Itanium Stack Structures 44
11.1.1 Itanium Memory Stacks 44
11.1.2 Itanium Register Stacks 44
11.2 Calling Procedures and Functions 45
11.2.1 Register Conventions 45
11.2.2 Call and Return Branch Instructions 45
11.2.3 Argument Passing 46
11.2.4 A Practical Example 46

12 Program Performance **50**
12.1 Processor-Level Parallelism . . . 50
12.2 Instruction-Level Parallelism . . . 51
12.3 Explicit Parallelism . . . 51
12.3.1 Instruction Templates . . . 51
12.3.2 Data Dependencies and Speculation . . . 52
12.3.3 Control Dependencies and Speculation . . . 54
12.4 Program Optimizations . . . 55
12.4.1 Performance Considerations . . . 55
12.4.2 Low-level Optimization Hints . . . 57
12.4.3 Performance Monitoring . . . 58
12.5 Loop Optimization as a Practical Example . . . 59
12.5.1 Loop Unrolling . . . 59
12.5.2 Software-Pipelined Loops . . . 59
12.5.3 Writing a Software Pipelined Loop . . . 61

13 Itanium Implementations **65**
13.1 The Itanium-Family Processors . . . 65
13.1.1 Cache Hierarchy . . . 65
13.1.2 Execution Units and Issue Ports . . . 66
13.1.3 Pipelines . . . 68
13.2 The Ski Simulator . . . 68

List of Figures

1 A slightly modified version of the SQUARES program from Evans and Trimper 15
2 The Itanium extract and deposit instructions 27
3 A slightly modified version of the DOTCLOOP program from Evans and Trimper 43
4 Passing arguments via registers and the memory stack 46
5 A slightly modified version of the BOOTH function from Evans and Trimper 48
6 A slightly modified version of the DECNUM3 program from Evans and Trimper 49
7 A slightly modified version of the DOTCTOP2 program from Evans and Trimper 64
8 Structure of the cache hierarchy for the Itanium processors 65

List of Tables

1	Size and numeric range (in decimal notation) of Itanium integer data types	3
2	IEEE floating-point representation	4
3	Itanium instruction types and the corresponding execution units	7
4	Itanium addressing modes and effective addresses	8
5	The names and uses of the Itanium general-purpose registers	10
6	The names and uses of the Itanium floating-point registers	11
7	The names and uses of the Itanium branch registers	11
8	The names and uses of the Itanium predicate registers	12
9	Comparison of the Itanium integer and floating-point instructions	28
10	Meanings of the special IEEE floating-point representations	28
11	Assembler mnemonics for the `fclass` instruction	35
12	The registers, and their uses, of the DECNUM3 stack frame	47
13	Itanium instruction templates	51
14	Characteristics of the current Itanium processors	66
15	Characteristics of the Itanium and Itanium 2 cache structures	67
16	Possible dual-issue instruction bundles for the Itanium and Itanium 2 processors	67
17	The Itanium and Itanium 2 pipelines	68

Preface

The Itanium family of processors represents Intel's foray into the world of Explicitly Parallel Instruction Computing and 64-bit system design. Within this survey is contained an introduction to the Itanium architecture and instruction set, as well as some of the available implementations. We have attempted to distill the relevant information from the thousands of pages of Itanium documentation and reference materials cited at the end of this work by taking a programmer's perspective.

This survey largely follows the structure, form, and content of an excellent book by James Evans and Gregory Trimper, entitled *Itanium Architecture for Programmers*. We have, of course, taken the liberty to rearrange the topics, omit the less important details, and expand the most relevant discussions with appropriate information from other sources; in other words, we do more than simply summarize the book. Nevertheless, we gratefully acknowledge the significant impact that their work has had on this survey.

We cover the following topics in varying levels of detail:

- the important characteristics of the Itanium architecture (Sections 1–3),
- programming with the Itanium instruction set (Sections 4–11),
- program performance factors and optimization techniques (Section 12), and
- several implementations of the Itanium architecture (Section 13).

It is not our intention to provide exhaustive discussions of the Itanium architecture, its instruction set, or any of the available implementations. We have made an effort to include those topics and details that we found most useful during our initial experimentation with the Itanium architecture. Likewise, where useful or important details have been omitted intentionally, due either to space and formatting constraints or to the intended scope of this work, we have made an effort to cite specific sections and pages within the reference materials that will enhance the included discussion.

Our hope is that this survey will serve as a practical introduction to creating new applications for the Itanium architecture.

Salt Lake City, Utah
August 2003

1 The Itanium Architecture

A computer system may be categorized in terms of two basic characteristics: its organization and its architecture. The *organization* of a computer system describes its overall structure and the elements of which the system is composed. In this survey, we are concerned with the details of Intel's Itanium architecture and not the more general topic of computer organization. Many instructive resources cover both of these topics simultaneously; in particular, we recommend the well-known texts by John Hennessy and David Patterson (see the section entitled "Bibliography and Additional Resources" on page 70).

The *architecture* of a computer system, on the other hand, describes the structure and operation of the system as visible to an assembly language programmer. A computer architecture is therefore an abstraction, consisting of the programming interface for controlling the operation of the system.

A related, but wholly distinct, concept is that of an *implementation*. An implementation is the realization of the structure and operation prescribed by a computer architecture using various hardware and software components. For example, the Itanium 2 processor is just one of the several available implementations of the Itanium architecture.

An Analogy from Evans and Trimper. In their book, *Itanium Architecture for Programmers*, James Evans and Gregory Trimper offer a useful analogy, based on pianos, to help clarify the distinction between an architecture and an implementation. We paraphrase their example and include it here:

A piano architecture is defined by the specification of the keyboard. The keyboard is the player's interface to the instrument, and it consists of 88 keys: 36 black keys and 52 white keys. Notes of various specified frequencies are sounded by striking a particular key. The size and arrangement of the keys are identical for all modern piano keyboards, so a person who can play the piano can play *any* piano.

Many implementations of the piano architecture are possible. Implementations may be distinguished by the types of materials used to construct the piano, by the size and shape of the instrument, or any number of other decisions made by a particular manufacturer concerning the details of the piano. Nevertheless, any piano player will be able to play the final product.

Likewise, a computer architecture specifies the programmer's interface for controlling the operation of the system. Many implementations of the computer architecture are possible, and they are distinguished by size, cost, and performance characteristics. However, any computer program that runs on one machine should run on *any* machine conforming to the same architecture.

Explicitly Parallel Instruction Computing. Modern computer architectures are generally classified as one of three types. *Complex Instruction Set Computers* (CISC) usually provide a large number of machine instructions, each of which may exhibit many different styles. CISC machines are often difficult to implement because each type of instruction may require a large portion of the available die area. In contrast, *Reduced Instruction Set Computers* (RISC) provide far fewer machine instructions and far fewer instruction styles. As a result, faster circuitry may be possible, and RISC programs may execute faster than their CISC counterparts, even though they are typically composed of a larger number of machine instructions.

A few, mostly experimental, RISC architectures employ *very long instruction words* (VLIW) to guide the simultaneous execution of several RISC-like instructions. In the past, the advantages of the VLIW approach were overshadowed by its disadvantages. Analyzing and implementing instruction-level parallelism required very sophisticated compilers, and accommodating the architectural latency among the instructions required that software programs be recompiled (and thus redistributed) for each new hardware implementation. However, after a thorough analysis by B. Rau, minimal modifications to the VLIW approach enabled the architecture-implementation difficulties to be overcome. These results lead directly to the third, and newest, class of modern computer architecture: *Explicitly Parallel Instruction Computer* (EPIC). Intel's Itanium architecture is the first EPIC design.

64-Bit Systems. A computer system may also be classified according to the width of its *datapath*. This width describes the number of bits that can "flow" through the computer's internal conduits in parallel.

Although EPIC architectures are a relatively recent development, processors built around a 64-bit datapath are not. The Alpha processor, marketed by Digital Equipment Corporation, was the first 64-bit RISC computer to find commercial success. Other manufacturers, such as Hewlett-Packard, have also marketed 64-bit systems with varying degrees of success.

The Itanium architecture is Intel's first 64-bit design. While it is too soon to declare the Itanium architecture an overwhelming success, we are hopeful that the implications of EPIC principles, when combined with a 64-bit design, will lead to a viable and affordable platform for building and running large-scale scientific and high-performance computing applications.

In the following pages, we discuss those aspects of the Itanium architecture that are most relevant to both high-level and assembly language programmers. If our hope is realized, you will be well-prepared to face the new programming challenges.

2 Instruction Set Architecture

Just as a computer architecture abstracts the structure and operation of a computer system, an *instruction set architecture* (ISA) abstracts the interface between a computer's hardware and lowest-level software. With an understanding of the ISA, a programmer knows, in principle, what the computer system can and cannot do and how to accomplish a given task efficiently.

Instruction sets may be classified according to the number of addresses contained within the typical instruction. Like most RISC designs, the Itanium architecture describes a two-address machine with respect to its load and store operations: one operand is a memory address and the other is a processor register. Most other Itanium instructions involve at least three addresses (for example, the `add r1=r2,r3` instruction) and therefore specify two source operands (`r2`, `r3`) and one destination operand (`r1`).

2.1 Information Units and Data Types

The basic unit of information in the Itanium architecture is the 8-bit *byte*. Unlike previous Intel architectures, the Itanium architecture assigns each byte a 64-bit address. The architecture also describes several multi-byte units that are composed of groups of adjacent bytes: the 16-bit *word* (2 bytes), the 32-bit *double word* (4 bytes), and the 64-bit *quad word* (8 bytes). Each of these multi-byte units is also addressable.

2.1.1 Integers

Although the Itanium load and store instructions are able to manipulate information units smaller than 64 bits in width, integer arithmetic instructions only operate on quad word data. Likewise, Itanium logical instructions only work with quad word data; these instructions, however, provide some access to the data at the bit or group-of-bits level. Table 1 expresses the size and numeric range of the available Itanium integer data types.

Unit	Bits	Bytes	Signed Integer	Unsigned Integer
Byte	8	1	-128 to +127	0 to 255
Word	16	2	-32,768 to +32,767	0 to 65,535
Double word	32	4	-2,147,483,648 to +2,147,483,647	0 to 4,294,967,295
Quad word	64	8	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	0 to 18,445,744,073,709,551,615

Table 1: Size and numeric range (in decimal notation) of Itanium integer data types

2.1.2 Floating-Point Numbers

In addition to integers, the Itanium architecture provides instructions for manipulating floating-point numbers. These numbers are typically represented by a *significand* that is multiplied by some power of two. An *exponent* and *sign* are also packed with the significand.

Historically, computer manufacturers defined their own formats for floating-point numbers. Only after the set of standards documented in ANSI/IEEE 754, *IEEE Standard for Binary Floating-Point Arithmetic*, emerged was there any agreement between manufactures concerning floating-point representation.

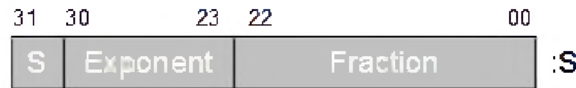
The standard defines four floating-point formats: single, double, extended single, and extended double. The former two formats have been supported by nearly every new architecture that has been developed in the time since the standard was defined, while the latter two offer flexibility for supporting older, proprietary floating-point representations. For the purposes of this survey, we only consider the widely supported IEEE

single- and double-precision floating-point numbers. Table 2 lists some important characteristics of each floating-point format that we consider here. Note that in IEEE representation, the significand consists of an implicit "hidden bit" followed by the fraction; this bit is suppressed to reduce storage requirements.

Characteristic	Single	Double
Size in memory		
Sign	1 bit	1 bit
Exponent	8 bits	11 bits
Fraction	23 bits	52 bits
Total	32 bits	64 bits
Bias for exponent	127	1023
Minimum magnitude	$1.175 * 10^{-38}$	$2.225 * 10^{-308}$
Maximum magnitude	$3.403 * 10^{+38}$	$1.798 * 10^{+308}$
Precision		
Binary	24 bits	53 bits
Decimal	6 digits	16 digits

Table 2: IEEE floating-point representation

Single precision. An IEEE single-precision floating-point number consists of four adjacent bytes in memory. In a little-endian representation the bits are labeled from right to left, as follows:

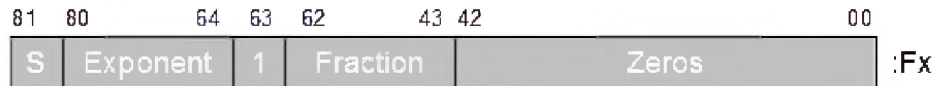


Generally, the value of the number is given by

$$(1 - 2 * S) * 1.F * 2^{(E-B)}$$

where S is the sign of the number (0 for positive, 1 for negative), F is the binary fraction, $1.F$ is the significand, E is the true exponent, and B is the bias. There are special cases when these bits are interpreted differently, for example, when manipulating an integer value stored in a floating-point register.

When an IEEE single-precision floating-point number is stored in an Itanium processor register, the regions are arranged as follows:



Note that the "hidden bit" in the IEEE representation is made explicit in an Itanium processor register.

Double precision. An IEEE double-precision floating-point number consists of eight adjacent bytes in memory. In a little-endian representation the bits are labeled from right to left, as follows:

Again, the value of the number is generally given by

$$(1 - 2 * S) * 1.F * 2^{(E-B)}$$

where S, F, $1.F$, E, and B are as before. There are again special cases when these bits are interpreted differently.

When an IEEE double-precision floating-point number is stored in an Itanium processor register, the regions are arranged as follows:

As with single-precision floating-point numbers, the hidden bit is made explicit in an Itanium processor register.

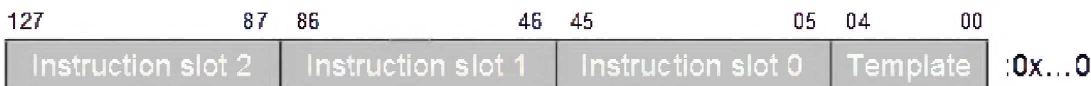
2.1.3 Alphanumeric Characters

Binary numbers can encode any information, including alphanumeric characters (letters, numerals, punctuation marks, etc.). Numerous encoding schemes exist, and the supported schemes are largely dependent upon which operating system and programming environment are used. For this reason, we omit any further discussion of this data type, except to say that the Itanium architecture features instructions for manipulating narrow information units (that is, bytes, words, and double words), many of which are discussed in later sections. Managing strings of alphanumeric data is relegated to the programmer or compiler.

We recommend that consult your system's documentation to see which encoding schemes and character sets are supported.

2.2 Instruction Formats

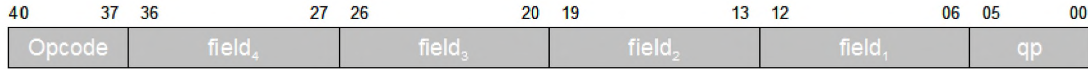
The Itanium architecture specifies a seemingly awkward 41-bit instruction width. While the rationale behind this choice is largely unimportant to the discussion at hand, it suffices to say that, in the final design, Itanium instructions are always fetched in groups of three and are packaged with a 5-bit *instruction template*, as follows:



This layout yields a 128-bit *instruction bundle*, where the template supplies additional information instructing the CPU how to decode and execute the three instructions. Instruction bundles are always treated as little-endian structures, as shown in the figure above, and are always 16-byte aligned; that is, the four lowest-numbered bits of a bundle's address are always zeros.

Instruction templates are one of 32 predefined bit patterns that describe the three instructions contained within the bundle. We defer any further discussion of these structures until Section 12.

While we are more concerned with the actual function of the Itanium instructions, it is useful to consider briefly the bit-field layout of a single Itanium instruction:



Itanium load and store operations require two operand specifiers (a source and a destination), and the arithmetic and logic operations require three operand specifiers (two sources and a destination). Some Itanium instructions have two destination operands, and so require four operand specifiers. Thus, the Itanium instruction layout may have as many as six main bit-fields.

The field labeled `qp` provides for a qualifying predicate, the fields labeled $\texttt{field}_1$ to $\texttt{field}_4$ provide for up to four operands, and the highest four bits specify the major opcode. Bits may be reinterpreted when an instruction requires less than four operands, or when numeric constants are packaged within the instruction itself as *immediate data*.

2.3 Instruction Classes

Itanium instructions can be divided into six basic classes:

- Type A instructions include standard arithmetic and logic operations on integers (add, multiply, Boolean AND, etc.), as well as comparison operations on data values.
- Type I instructions include other operations on integer data types, for example, bit-shifting, moving data to and from special purpose registers, and multimedia instructions.
- Type M instructions include the load and store operations for both integer and floating-point data, the operations for moving data between general-purpose integer registers and floating-point registers, and the instructions that give the programmer a limited degree of control over the system's memory hierarchy.
- Type B instructions include the branching and jumping operations, as well as those for calling and returning from functions or procedures.
- Type F instructions include those operations on floating-point data that are not Type M instructions.
- Type X instructions include a few special Itanium instructions that encode more information than would normally fit into the 41-bit instruction width. These instructions consume two slots in an instruction bundle.

Table 3 (next page) shows the Itanium instruction types and the execution units that actually perform the operations. Not surprisingly, there is a correspondence between the I, M, B, and F instructions and the I, M, B, and F execution units that decode and execute them. Type A instructions, which are the most common, can be executed by both I- and M-units, providing the potential for a high degree of instruction-level parallelism.

An implementation of the Itanium architecture may include more than one of each type of execution unit. As an example, the Itanium 2 processor includes four M-units (two load, two store) and two I-units, but only one F-unit.

Operation	Instruction Type	Execution Unit
Arithmetic, logic, comparison	A	any available I- or M-unit
Other integer operations	I	I-unit
Memory access and data movement	M	M-unit
Branches and calls	B	B-unit
Floating-point operations	F	F-unit
Special two-slot instructions	X	I- or B-unit, depending on operation

Table 3: Itanium instruction types and the corresponding execution units

2.4 Addressing Modes

The Itanium ISA supports four addressing modes: immediate, register direct, register indirect, and autoincrement. Table 4 (next page) captures the important characteristics of the available Itanium addressing modes, each of which we describe fully below.

2.4.1 Immediate Addressing

When *immediate addressing* is used, the instruction itself contains the operand data. Because the data is already in the CPU, no additional address calculations or memory fetches are required.

We have already encountered immediate addressing briefly: numeric constants whose values are known at the time of program assembly or compilation can packaged within the bit-field of a given instruction. Immediate addressing is almost always used for these sorts of operands. Also, you will recognize that immediate addressing is useful only for source, and not destination, operands.

2.4.2 Register Direct Addressing

An instruction may contain an address that points to the operand data; this addressing mode is called *direct addressing*. The Itanium ISA is a register-to-register architecture and allows only the load and store instructions to operate on data in memory. Thus, only *register direct addressing*, where the bits within the instruction specify the "address" (name or number) of a processor register that contains the operand data, is permitted by the Itanium architecture.

2.4.3 Register Indirect Addressing

An instruction may also contain an address pointer to the operand data; this addressing mode is called *indirect addressing*. The bits within the instruction contain the register address, say `rX`. When the instruction executes, the contents of this register `rX` interpreted as the *effective address* of the information unit containing the actual data. For the Itanium architecture, this two-phase addressing mode is more strictly called *register indirect addressing*.

2.4.4 Autoincrement Addressing

Often it is useful to refer to operand data using register indirect addressing and then adjust the address contained within that register to point to the next identically sized information unit. Stepping through an array of data is an example of a common task where this addressing mode proves useful.

The Itanium ISA supports this capability with its *autoincrement addressing* mode. The postincrement value is not limited to the size of particular data types. For store operations, the value is expressed as a 9-bit

Addressing Mode	Assembler Syntax	Effective Address
Immediate	`imm`	Bits packaged within the instruction are interpreted as an integer value, typically signed, or as an instruction "subcode" that is used to select specific cases of the instruction
Register Direct	`rX`	The named register
Register Indirect	`[rX]`	Contents of the named register
Autoincrement	`[rX], imm` or `[rX], rY`	Contents of the named register; the register value is then postincremented by the signed quantity given statically as `imm` (load and store operations) or dynamically in register `rY` (load operations only)

Table 4: Itanium addressing modes and effective addresses

signed immediate constant within the instruction. For load operations, it can also be specified dynamically using a value in an Itanium general-purpose register.

Specifying the postincrement value as a signed constant allows the programmer to step through an array in either direction, depending on whether the register points to the first or last data element.

3 Architectural Registers

The Itanium architecture includes an unprecedented number of registers, including an instruction pointer, 128 general-purpose registers, 128 floating-point registers, 8 branch registers, 64 predicate registers, as many as 128 special-purpose (application) registers, various system information registers, and many others. Such a large number of registers enables numerous computations to be performed without the need to repeatedly spill and fill intermediate results to memory.

The Itanium registers vary greatly in their size, features, and uses. Following the nomenclature used by Evans and Trimper, we characterize the Itanium registers with the following terms:

- A register is *constant* if its value has been permanently defined at the hardware level.
- A register is *special* if it has some purpose assigned to it, either at the hardware level or by software convention.
- A register is *scratch* if it may be used freely by a procedure or function at any calling level; the caller must save any important contents of these registers.
- A register is *preserved* if a calling routine depends on its contents; any called procedure must save and restore the contents of these registers for its caller.
- A register is *automatic* if its name only has a dynamic correspondence to a physical register; these registers are automatically spilled to and filled from memory during allocation by the hardware, as necessary.
- A register is *read-only* if its value is dynamically maintained at the hardware level or by the operating system; read-only registers cannot be modified by an application program.

Please refer to these descriptions as we detail the Itanium architectural registers.

3.1 Instruction Pointer

The Itanium instruction pointer (IP) supports the instruction fetch cycle; it points to the currently executing instruction bundle. In most other architectures, this register is called the program counter. The Itanium IP is 64 bits wide and can accommodate full address pointers:

Itanium instructions are always fetched three at a time, as 128-bit instruction bundles. The lowest four bits of the IP are, therefore, always zero.

3.2 General-Purpose Registers

The Itanium architecture specifies 128 general-purpose registers, named Gr_0–Gr_{127}. Each of these registers is 64 bits wide and can accommodate both full address pointers and either signed or unsigned integers:

Each general register has an associated 65th bit, called the Not a Thing (NaT) bit. This bit is used to indicate whether the value stored in a register is valid. When the contents of a marked register are used by

an operation, the NaT bit of the destination register will automatically be set. The invalid condition may be carried through a sequence of instructions, to be dealt with when convenient. NaT bits are important for software that utilizes speculative loads.

Table 5 lists the names and standardized uses of the Itanium general-purpose registers.

Register	Assembler Name	Other Name	Class	Notes
Gr_0	`r0`		Constant	Always contains 0; writes are illegal
Gr_1	`r1`	`gp`	Special	Global data pointer
Gr_2, Gr_3	`r2, r3`		Scratch	Often useful with `addl` instruction
Gr_4–Gr_7	`r4–r7`		Preserved	
Gr_8–Gr_{11}	`r8–r11`	`ret0–ret3`	Scratch	Integer values returned by a function
Gr_{12}	`r12`	`sp`	Special	Stack pointer (always modulo 16)
Gr_{13}	`r13`	`tp`	Special	Thread pointer (requires operating system support)
Gr_{14}–Gr_{31}	`r14–r31`		Scratch	
Gr_{32}–Gr_{39}	`r32–r39`	`in0–in7`	Automatic	Up to 8 input arguments to a function
Gr_{32}–Gr_{127}	`r32–r127`		Automatic	Stacked input registers; safe
Gr_{32}–Gr_{127}	`r32–r127`	`loc0–loc95`	Automatic	Stacked local registers; safe
Gr_{32}–Gr_{127}	`r32–r127`	`out0–out95`	Automatic	Stacked output registers
Gr_{32}–Gr_{127}	`r32–r127`		Automatic	Rotating registers (groups of 8); they overwrite the stacked registers of the current procedure

Table 5: The names and uses of the Itanium general-purpose registers

3.3 Floating-Point Registers

The Itanium architecture specifies 128 floating-point registers, named Fr_0–Fr_{127}. Each of these registers is 82 bits wide and can accommodate expanded forms of IEEE single- or double-precision floating-point values, as well as signed or unsigned 64-bit integers:

The floating-point registers do not have an associated invalidity bit; rather, a special value called Not a Thing Value (NaTVal) indicates whether the contents of a floating-point register are valid.

Table 6 (next page) lists the names and standardized uses of the Itanium floating-point registers.

Register	Assembler Name	Other Name	Class	Notes
Fr_0	`f0`		Constant	Always +0.0; writes are illegal
Fr_1	`f1`		Constant	Always +1.0; writes are illegal
Fr_2–Fr_5	`f2–f5`		Preserved	
Fr_6–Fr_7	`f6–f7`		Scratch	
Fr_8–Fr_{15}	`f8–f15`		Scratch	Floating-point arguments to a function and values return by a function
Fr_{16}–Fr_{31}	`f16–f31`		Preserved	
Fr_{32}–Fr_{127}	`f32–f127`		Scratch	Rotating registers

Table 6: The names and uses of the Itanium floating-point registers

3.4 Branch Registers

The Itanium architecture specifies eight branch registers, named Br_0–Br_7. Each of these registers is 64 bits wide and can therefore accommodate full address pointers:

Itanium instructions are always fetched three at a time, as 128-bit instruction bundles. The lowest four bits of a branch register are, therefore, always zero.

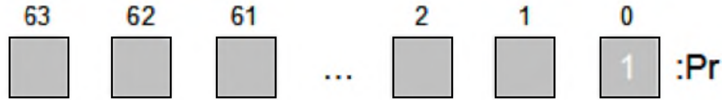
Branch registers specify the target address of indirect branches. Table 7 lists the names and standardized uses of the Itanium branch registers.

Register	Assembler Name	Other Name	Class	Notes
Br_0	`b0`	`rp`	Scratch	Return link
Br_1–Br_5	`b1–b5`		Preserved	
Br_6–Br_7	`b6–b7`		Scratch	

Table 7: The names and uses of the Itanium branch registers

3.5 Predicate Registers

The Itanium architecture specifies 64 predicate registers, named Pr_0–Pr_{63}. Each of these registers is only one bit wide and can therefore accommodate either a Boolean true (1) or false (0) value:



These registers control the conditional execution of instructions and conditional branches. Table 8 lists the names and standardized uses of the Itanium predicate registers. Each bit in the 64-bit predicate vector is individually addressable. No predicate registers are automatically stacked at the time of a procedure call; if necessary, the entire predicate vector may be saved to a general-purpose register.

Register	Assembler Name	Other Name	Class	Notes
Pr_0	`p0`		Constant	Always true (1); writes are discarded
Pr_1–Pr_5	`p1–p5`		Preserved	Fixed; safe
Pr_6–Pr_{15}	`p6–p15`		Scratch	Fixed; unsafe
Pr_{16}–Pr_{63}	`p16–p63`	`pr.rot`	Preserved	Rotating registers

Table 8: The names and uses of the Itanium predicate registers

3.6 Application Registers

The Itanium architecture allows for as many as 128 application registers (named Ar_0–Ar_{127}) to be defined. These 64-bit registers can accommodate full address pointers and either signed or unsigned integers:

Application registers perform specific tasks associated with various instructions in an application-level program. We omit any further details of the Itanium application registers here. Consult the Intel Itanium architecture documentation for more information.

3.7 System Information Registers

The Itanium architecture also includes registers that provide information concerning the hardware implementation to application programs.

For example, an application can determine implementation-dependent features, such as the processor's manufacturer or its family and model numbers, by reading the `cpuid` (processor identification) registers.

The Itanium architecture also allows for as many as 256 64-bit `pmd` (performance monitor data) registers. These registers record certain aspects of the system's performance that can be used for tuning applications. An operating system may allow read-only access to these registers by application-level code. The Itanium architecture requires that, at a minimum, eight `pmd` registers be implemented.

3.8 Other Processor Registers

In addition to those we have discussed, the Itanium architecture includes a number of other registers:

- Various state management registers, like the `ar.pfs` (previous function state) register, where prior state information can be preserved in hardware rather than a slower memory stack.
- Various system control registers, like the `psr` (processor status) register, where the operating system and hardware can track critical aspects of machine state.
- Several more, highly specialized registers that generally require privileged instructions to access them.

For more details concerning these other Itanium registers, we recommend the Intel Itanium architecture documentation.

4 Itanium Assembler Statements

Before diving into a detailed discussion of the available instructions, we introduce the basic syntax of Itanium assembler statements:

```
[label:]  [(qp)]  mnemonic[.comp]  dst=src[;;]  [// comment]
```

where `[ ]` denotes an optional syntactical element and:

- `label` is a symbolic address in the form of a character string terminated by a colon (`:`),
- `(qp)` specifies a qualifying predicate register,
- `mnemonic` specifies a name that uniquely identifies an Itanium instruction,
- `comp` specifies one or more *instruction completers* to indicate variations on a base instruction mnemonic,
- `dst` specifies the destination operand(s),
- `src` specifies the source operand(s),
- `;;` is an *explicit stop* used to identify Itanium instruction bundles or data dependencies, and
- `// comment` is a human-language description of the assembler statement.

While not altogether void of traits in common with assembly statements for other architectures, many of these elements are unique to the Itanium architecture.

Typically, each line in an assembly language program is one statement that may be imperative, declarative, or controlling:

- *Imperative statements* represent machine instructions in symbolic form. These statements are the most common type.
- *Declarative statements* control the allocation of memory or perform naming functions. These statements do not generate machine instructions to be executed at runtime; rather, they set aside space, define symbols, or initialize the contents of particular memory locations.
- *Controlling statements* give the programmer a limited degree of control over certain aspects of the assembly process.

To illustrate the Itanium assembly language, we include a (slightly modified) example program from Evans and Trimper called `SQUARES`. The program populates a table in memory with the squares of the first three integers using tabular differences. The `SQUARES` code is given in Figure 1 (next page).

Even without any knowledge of particular Itanium instructions, you should be able to recognize many elements of the Itanium architecture that we have already discussed: various processor registers, like the general-purpose registers Gr_{20}–Gr_{22} and the branch register Br_0, or perhaps the use of register indirect addressing, as in `st8 [r14]=r20;;`. You will see, too, that the `SQUARES` program includes imperative, declarative, and controlling assembler statements.

While `SQUARES` is a trivial example and is not particularly useful except for illustrative purposes, the program sets the stage nicely for our discussion of the Itanium instruction set.

```
// SQUARES: Populate a table of squares

        .data                           // Declare data section
        .align   8                      // Specify desired alignment
sq1:    .skip    8                      // To store 1 squared
sq2:    .skip    8                      // To store 2 squared
sq3:    .skip    8                      // To store 3 squared

        .text                           // Declare code section
        .align   32                     // Specify desired alignment
        .global  main                   // Mark mandatory program entry
        .proc    main
main:
        .body                           // Begin procedure 'main'
first:  mov      r21=1;;                // Gr21 = 1st tabular difference
        mov      r22=2;;                // Gr22 = 2nd tabular difference
        mov      r20=1;;                // Gr20 = 1st square
        addl     r14=@gprel(sq1),gp;;   // Point to storage for 1st square
        st8      [r14]=r20;;            // Store 1st square

        add      r21=r22,r21;;          // Adjust 1st tabular difference
        add      r20=r21,r20;;          // Gr20 = 2nd square
        addl     r14=@gprel(sq2),gp;;   // Point to storage for 2nd square
        st8      [r14]=r20;;            // Store 2nd square

        add      r21=r22,r21;;          // Adjust 1st tabular difference
        add      r20=r21,r20;;          // Gr20 = 3rd square
        addl     r14=@gprel(sq3),gp;;   // Point to storage for 3rd square
        st8      [r14]=r20;;            // Store 3rd square

done:   mov      r8=0;;                 // Signal completion
        br.ret.sptk.many b0;;           // Return to command line
        .endp    main                   // End procedure `main`
```

Figure 1: A slightly modified version of the `SQUARES` program from Evans and Trimper

5 Integer Instructions

A large number of computer programs run simply by manipulating integer data types. Many integer operations are provided by the Itanium ISA, so we being our discussion of the architecture's instruction set with these instructions.

5.1 Arithmetic Instructions

The Itanium integer arithmetic instructions include addition, subtraction, bit-shift left with addition, and parallel multiplication of 16-bit values. These Type A instructions are among the most commonly used.

5.1.1 Addition

Several forms of the Itanium addition instruction are available:

```
add     r1=r2,r3         // r1 <- r2 + r3
add     r1=r2,r3,1       // r1 <- r2 + r3 + 1
adds    r1=imm14,r3      // r1 <- sext(imm14) + r3
addl    r1=imm22,r3      // r1 <- sext(imm22) + r3
add     r1=imm,r3        // r1 <- sext(imm) + r3
```

where `sext` denotes that the immediate constant is sign-extended to 64 bits before being used in the operation. The register designations `r1`, `r2`, and `r3` refer to the particular encoding found in fields in the bit layout of an instruction. Any one of the Itanium general-purpose registers Gr_0–Gr_{127} may be specified with this instruction; however, only Gr_0–Gr_3 may be used with the `addl` form.

The last form of the Itanium `add` instruction is an example of an *assembler pseudo-op*. A pseudo-op is a convenient form of an instruction that the assembler will "transform" according to context. For example, if the constant `imm` can be represented as a two's complement integer using 14 or fewer bits, the assembler will generate the appropriate `adds` instruction. On the other hand, if the representation of that constant requires more than 14 bits, the appropriate `addl` instruction will be generated. Note that in this case, the choice of registers for the second source operand is constrained to one of Gr_0–Gr_3, as mentioned above.

5.1.2 Subtraction

Fewer forms of the Itanium subtraction instruction are available:

```
sub    r1=r2,r3        // r1 <- r2 - r3
sub    r1=r2,r3,1      // r1 <- r2 - r3 - 1
sub    r1=imm8,r3      // r1 <- sext(imm8) - r3
```

where the notational conventions are the same as for addition. Note that only one, relatively narrow, representation can be used for the immediate constant.

5.1.3 Shift Left and Add

A third integer arithmetic operation is available, and it combines a bit-shift left with addition, as follows:

```
shladd    r1=r2,count,r3      // r1 <- 2^count * r2 + r3
```

where `count` specifies the number of bit positions, ranging from a minimum of one to a maximum of four, that the value in first source register will be shifted to the left.

Special cases of integer multiplication. When Gr_0 is specified as the second source register, the `shladd` instruction can compute 2, 4, 8, or 16 times the value contained in the first source register, depending on the value of `count`. When the two source registers are the same general-purpose register, say Gr_n, this instruction can compute 3, 5, 9, or 17 times the value contained in Gr_n, again, depending on the value of `count`.

Array indexing. Computing the address of a particular element in an array can be completed in one step with the `shladd` instruction:

address = (*element index*) ∗ (*size of data type*) + (*starting address of array*)

becomes

```
shladd   addr=index,count,array_addr
```

where `addr` is the general-purpose register used to hold the element's computed address, `index` is a general-purpose register holding the element number (zero-based indexing), `count` corresponds to the size of the individual array elements, and `array_addr` is a general-purpose register holding the starting address of the array. With the `shladd` instruction, it is possible to work with the whole array using only two general-purpose registers: one containing the array's starting address and one containing the current element number.

5.1.4 Multiplication and Division of 64-bit Integers

A multiply or divide instruction requires more stages to implement at the digital logic level than a simple addition or subtraction instruction. This requirement implies that multiplication and division instructions will take longer to execute than other instructions. RISC and EPIC architectures strive to make instruction execution times as consistent as possible across the entire instruction set. As a result, the Itanium architecture does not provide instructions for full width integer multiplication or division.

The Itanium ISA does include a special instruction that will perform integer multiplication using the floating-point registers. We discuss this topic further in Section 8. No such instruction is available for integer division, but virtually all programming environments provide some form of a software substitute. Consult your system's documentation to see which (possibly unpublished) internal routine or inline instruction sequence is used.

5.1.5 Multiplication of 16-bit Integers

The Itanium ISA includes two forms of a *parallel* instruction that multiplies two 16-bit signed integer pairs and produces two independent 32-bit signed integer products:

```
pmpy2.l   r1=r2,r3     // parallel multiply, left form
pmpy2.r   r1=r2,r3     // parallel multiply, right form
```

With the left form, the result of multiplying bits <63:48> of each source register is placed in bits <63:32> of the destination register, while the result of multiplying bits <31:16> of both sources is placed in <31:0> of the destination.

In contrast, with the right form, the result of multiplying bits <47:32> of each source register is placed in bits <63:32> of the destination register, while the result of multiplying bits <15:0> of both sources is placed in <31:0> of the destination.

The Itanium ISA provides a number of other parallel operations; these instructions are introduced in Section 9.

5.1.6 Special-Case Arithmetic Operations

We saw earlier that the Itanium assembler provides a pseudo-op for the `add` instruction with an immediate constant. Assembler pseudo-ops may be provided for very common or useful operations, largely as a convenience to the programmer.

An ISA will typically provide instructions for general operations that include the more common operations as special-cases. Some architectures may provide machine instructions for the special cases, others may provide pseudo-ops, and others still may not provide either. Here, we discuss some common operations for which the Itanium ISA does not provide actual machine instructions but that can be written as special cases of more general instructions or as assembler pseudo-ops.

Negation. Many architectures include an operation for arithmetic negation of integers in two's complement notation. The Itanium ISA, however, includes neither a machine instruction nor an assembler pseudo-op to accomplish this task. Arithmetic negation can be accomplished using either one of two special cases of subtraction:

```
sub    r1=0,r3        // r1 <- 0 - r3 = -r3
sub    r1=r0,r3       // r1 <- r0 - r3 = 0 - r3 = -r3
```

With either form, the value contained in `r3` is subtracted from zero, and `r1` will contain the negated value.

Complementation. Likewise, the Itanium ISA does not include an instruction for computing the one's complement (bitwise complement) of a value. Again, either one of two special cases of subtraction can be used to accomplish this common task:

```
sub    r1=-1,r3         // r1 <- -1 - r3
sub    r1=r0,r3,1       // r1 <- r0 - r3 - 1 = 0 - r3 - 1 = -1 - r3
```

Having examined both negation and complementation, it should now be clear why the `sub` instruction syntax specifies that the value in register `r3` be subtracted from the immediate constant.

Copying. The `mov` instruction used in the `SQUARES` program is actually an assembler pseudo-op. The Itanium ISA lacks a machine instruction that moves data between general-purpose registers or that moves a constant value into a register. However, the assembler recognizes the `mov` pseudo-op and implements two forms:

```
mov    r1=imm22    becomes   addl    r1=imm22,r0
mov    r1=r3       becomes   adds    r1=0,r3
```

The second form results in having copies of the same data value in both registers.

Clearing. The Itanium ISA also lacks an instruction to clear the contents of a general-purpose register. However, any one of several other instructions will suffice:

```
mov       r1=0                // becomes adds r1=0,r0
sub       r1=rn,rn            // r1 <- rn - rn = 0
shladd    r1=r0,count,r0      // the value of count is irrelevant
```

In all cases, register `r1` will contain zero.

5.2 Data Access Instructions

Most modern computer designs include *cache structures* that attempt to reduce the time required to access data stored in memory. In some designs, the presence and type of cache are only matters of concern for the implementation, and the ISA will not include instructions for interacting with the cache structures. In other designs, the presence and type of cache are matters of concern for both the architecture and the implementation. In this case, the ISA may include instructions for influencing the behavior of the cache structures.

The Itanium architectures specifies that the cache structures be explicitly visible to the assembly language programmer. The Itanium ISA includes instructions for prefetching a line of data that will soon be needed by a program into the cache and for flushing a line of data that is no longer needed back to memory. We defer any further discussion of the Itanium cache structures until Section 13.

At the moment, our concern with the cache structures involves the instruction completers that can be used with the integer load and store operations to provide *hints* to these structures. We discuss the load and store instructions below.

5.2.1 Load Instructions

The Itanium ISA includes two forms of the integer load instruction:

```
ldsz.ldtype.ldhint    r1=[r3]          // r1 <- mem[r3]
ldsz.ldtype.ldhint    r1=[r3],r2       // r1 <- mem[r3]
                                       // r3 <- r3 + r2
ldsz.ldtype.ldhint    r1=[r3],imm9     // r1 <- mem[r3]
                                       // r3 <- r3 + sext(imm9)

ld8.fill.ldhint       r1=[r3]          // fill data and NaT bit
ld8.fill.ldhint       r1=[r3],r2       // fill data and NaT bit
                                       // r3 <- r3 + r2
ld8.fill.ldhint       r1=[r3],imm9     // fill data and NaT bit
                                       // r3 <- r3 + sext(imm9)
```

where *sz* is the size of the information unit in memory at the location specified by register `r3` from which 1, 2, 4, or 8 bytes are to be copied into the lowest-order 1, 2, 4, or 8 bytes of register `r1`. The loaded data is zero-extended to the full width of the register, as necessary. Note that the load instructions use register indirect addressing for the source operand and register direct addressing for the destination operand.

Several valid values for *ldtype*, the load type completer, are available. If this instruction completer is omitted, then an ordinary load is executed. Some of the other values can be used to indicated ordered, biased, speculative, and/or advanced loads and are discussed in Section 12.

There are three valid values for *ldhint*, the load hint completer: none, `nt1`, and `nta`. None, which corresponds to omitting the load hint completer, indicates an ordinary load operation; the processor hardware then assumes that the program associates temporal locality in the L1 cache with the loaded value. `nt1` provides the hint that the program considers the loaded value to have nontemporal locality in just the L1 cache, while `nta` hints that the program considers the loaded value to have nontemporal locality in all levels of the memory hierarchy. Using the load hint completers may avoid knocking out of the cache structures data that might be reused.

The load instructions provide for postmodification of the pointer value in register `r3` by a full 64-bit signed value stored in register `r2` or by a 9-bit signed constant, with values ranging from -256 to +255.

Finally, the `fill` form of this instruction loads 8 bytes and the NaT bit associated with register `r1`. This form is used to restore register contents when an operating system switches process contexts or when an application uses a preserved register.

5.2.2 Store Instructions

The Itanium ISA includes two forms of the integer store instruction:

```
stsz.sttype.sthint    [r3]=r2           // mem[r3] <- r2
stsz.sttype.sthint    [r3]=r2,imm9      // mem[r3] <- r2
                                        // r3 <- r3 + sext(imm9)

st8.spill.sthint      [r3]=r2           // spill data and NaT bit
st8.spill.sthint      [r3]=r2,imm9      // spill data and NaT bit
                                        // r3 <- r3 + sext(imm9)
```

where *sz* is the size of information unit in memory into which the lowest-order 1, 2, 4, or 8 bytes of the quantity in register `r2` are to be copied to the memory address specified in register `r3`. Note that the store instruction uses register direct addressing for the source operand and register indirect addressing for the destination operand.

There are two valid values for *sttype*, the store type completer: none and `rel`. None, which corresponds to omitting the store type completer, indicates an ordinary store operation. We omit any discussion of the `rel` store type completer but recommend the Itanium architecture documentation for further details.

There are two valid values for *sthint*, the store hint completer: none and `nta`. None, which corresponds to omitting the store hint completer, indicates an ordinary store operation; the processor hardware then assumes that the program associates temporal locality in the L1 cache with the stored value. `nta` provides the hint that the program considers the stored value to have nontemporal locality at all levels of the memory hierarchy. The use of `nta` may avoid knocking out of the cache structures data that might be reused.

The store instructions provide for postmodification of the pointer value in register `r3` by a 9-bit signed constant, with values ranging from -256 to +255.

Finally, the `spill` form of this instruction stores 8 bytes and the validity bit associated with register `r2`. This form is used to save register contents when an operating system switches process contexts or when an application uses a preserved register.

5.2.3 Move Long Immediate Instruction

The Itanium ISA provides a special instruction, called `movl`, that can accommodate a 64-bit immediate value:

```
movl    r1=imm64      // r1 <- imm64
movl    r1=label      // r1 <- 64-bit address for label
```

The 64-bit immediate value, or the full 64-bit address of `label` (determined by the linker), is copied into the general-purpose register `r1`. The `movl` instruction can use any general-purpose register in the range Gr_1–Gr_{127}, unlike the `addl` instruction we discussed previously. It is often used to establish pointers for subsequent load and store operations. Note that `movl` occupies two slots in an Itanium instruction bundle as a result of the 64-bit immediate value and is therefore one of the few Type X instructions.

5.2.4 Accessing Specialized Registers

Many of the Itanium's specialized registers contain information that is useful to an application-level program. Three `mov` assembler pseudo-ops provide the ability to copy values between specialized registers and general-purpose registers:

```
mov    r1=reg        // r1 <- contents of reg
mov    reg=r2        // reg <- contents of r2
mov    reg=imm8      // reg <- sext(imm8)
```

Note that the Itanium IP can be read, but not modified, using a `mov` instruction. Also, the last form, which uses an 8-bit immediate value as the source operand, can only be employed when the destination is one of the Itanium application registers (Ar_0–Ar_{127}).

5.3 Miscellaneous Integer Instructions

While most of the Itanium integer instructions operate on full 64-bit data, we have seen that the load and store operations can also manipulate narrower information units. The load instruction automatically zero-extends data that is less than 64 bits wide, but the Itanium ISA also includes a separate instruction to zero-extended data in a register to the full width. A similar instruction is provided for sign-extension.

5.3.1 Zero-Extend Instruction

Bit-masking is often used to force some bits of a register value to zero. This task can be accomplished with a Boolean AND operation. The Itanium architecture also includes an instruction to zero-extend a value to the full width of a register:

```
zxtxsz    r1=r3       // r1 <- zext(r3)
```

where `xsz` is 1, 2, or 4 to select the range of bit positions (<63:8>, <63:16>, or <63:32>) in the destination register `r1` that will be set to zero. The lowest-order 1, 2, or 4 bytes are copied from the source register `r3`.

A full width load followed by zero-extension is preferable to using the narrow load instructions when accessing individual bytes in memory is slower than this instruction pair. Which method is faster will typically depend on the implementation; nevertheless, either instruction sequence will produce the desired result on any Itanium implementation.

5.3.2 Sign-Extend Instruction

Similarly, the Itanium ISA includes an instruction that sign-extends a value to the full width of a register:

```
sxtxsz    r1=r3       // r1 <- sext(r3)
```

where `xsz` is 1, 2, or 4 to select the bit position (7, 15, or 31) in the source register `r3` that will be propagated as the sign bit in the destination register `r1`. Sign-extension is useful for constructing signed quantities from small information units that have been loaded by a narrow load instruction.

5.3.3 Instructions for Narrow Data Types

If the full numeric precision or large address space of the Itanium architecture are not necessary, the effectiveness of the Itanium's cache structures can actually improve if 32-bit address pointers and narrow information units are used. The Itanium provides arithmetic instructions for smaller data widths:

- Several parallel instructions, like the `pmpy2` instruction that we have already encountered, that operate on multiple narrow integer values simultaneously using the Itanium's 64-bit datapath; and
- `addp4` and `shladdp4`, which produce 64-bit address pointers from 32-bit addresses; these instructions are useful for migrating 32-bit code.

We discuss the parallel operations in Section 9 but omit any further details of the other narrow integer instructions. Consult the Itanium architecture documentation for more information.

6 Comparison and Branching Instructions

The power and flexibility of computer programming lies in the ability to control the logical flow of execution based upon currently calculated conditions. We now consider the features of the Itanium architecture that enable programmers to control the program's flow of execution.

6.1 Comparison Instructions

The Boolean true or false outcome of a comparison operation is typically used to choose between two alternative code sequences. Comparison operations are thus intimately tied to the concept of *predication*, where one set of actions is executed if a given premise is true and a different set is completed if that premise is false. The Itanium architecture supports predication more fully than any previous architecture. The Itanium predicate registers, which we introduced in Section 3, can capture the Boolean true or false result of a comparison operation and thus "control" which statements execute.

The Itanium ISA includes a number of 32- and 64-bit integer, double-precision floating-point, and parallel comparison instructions. We discuss the more common integer versions here; Section 8 and Section 9 cover the remaining versions.

6.1.1 Signed Comparison

There are six useful cases for comparing two values: equal (=), not equal (!=), less than (<), less than or equal (<=), greater than or equal (>=), and greater than (>).

Two versions of the signed comparison instruction are supported: `cmp`, for 64-bit quad word data values, and `cmp4`, for 32-bit double word data values. We describe only the syntax for the 64-bit instruction; the 32-bit `cmp4` operates in exactly the same manner.

Several forms of the instruction are available:

```
cmp.crel.ctype   p1,p2=r2,r3        // two registers
cmp.crel.ctype   p1,p2=imm8,r3      // immediate and one register
cmp.crel.ctype   p1,p2=r0,r3        // compare 0 to one register
cmp.crel.ctype   p1,p2=r3,r0        // compare one register to 0
```

where two predicate registers (Pr_0–Pr_{63}) must always be specified for `p1` and `p2`. Pr_0, which is always true, may be used in either position.

Typically, the comparison statements are read from left to right: In the two-register form, for example, `p1` is set to true and `p2` to false if `r2 crel r3` is true, and vice versa if the comparison is false.

There are six valid values for *`crel`*, the comparison relationship completer, each of which corresponds to one of the six comparison cases described above: `eq`, `ne`, `lt`, `le`, `ge`, and `gt`.

Several valid values for *`ctype`*, the comparison type completer, exist: none, `unc`, `or`, `and`, `or.andcm`, `orcm`, `andcm`, and `and.orcm`. None, which corresponds to omitting the comparison type completer, indicates an ordinary comparison operation, like the one described above. We discuss the unconditional comparison type completer `unc` momentarily. The remaining completers are used with the parallel comparison operations; parallel operations are introduced in Section 9.

6.1.2 Unsigned Comparison

When comparing two unsigned quantities for equality, the same instructions used for signed values will suffice: If the two bit patterns match at all bit positions, then the quantities are equal; otherwise, the quantities are not equal.

However, because of the two's complement representation used by most binary computers, the signed versions of the comparison instructions will not work for the remaining cases. The Itanium ISA provides four additional *`crel`* completers for dealing with unsigned values: `ltu`, `leu`, `geu`, and `gtu`.

6.1.3 Unconditional Comparison

We introduced the unconditional comparison type completer `unc` above. Valid forms of the compare instruction using this completer include:

```
cmp.crel.unc    p1,p2=r2,r3        // two registers
cmp.crel.unc    p1,p2=imm8,r3      // immediate and one register
cmp.crel.unc    p1,p2=r0,r3        // compare 0 to one register
cmp.crel.unc    p1,p2=r3,r0        // compare one register to 0
```

where the valid *crel* values are as before.

An ordinary comparison instruction executes and sets both predicate registers according to the comparative test when predicated true, but does nothing at all when predicated false. In the latter case, the values contained in the predicate registers will not change.

The unconditional comparison instruction behaves in the exact same manner when predicated true. However, if predicated false, this form sets the values in both predicate registers to false without actually performing a comparison. This form of the instruction is useful for constructing nested `if...then...else` structures, as we show in Section 10.

6.2 Branch Instructions

The Itanium ISA provides numerous branching abilities, among them the simple conditional and unconditional branch types. Five forms involving several instruction completers are available:

```
(qp)  br.brtype.bwh.ph.dh    target25     // relative to IP
(qp)  br.brtype.bwh.ph.dh    b2           // indirect addressing
      br.ph.dh               target25     // unconditional (pseudo-op)
      br.ph.dh               b2           // unconditional (pseudo-op)
(qp)  brl.brtype.bwh.ph.dh   target64     // relative to IP
```

where `qp` specifies the qualifying predicate register.

The branch target address can be specified using IP-relative addressing or indirect addressing with the branch register `b2`. If IP-relative addressing is used, the programmer can specify the address of the target instruction bundle using a symbolic label. The compiler or assembler will compute the appropriate 25-bit signed offset as `target25` − IP, resulting in a branch range of 2^{24} bytes (2^{20} instruction bundles).

To execute longer-range jumps, it is possible to load a full 64-bit address into one of the eight branch registers (Br_0–Br_7) and then specify that register as the branch target. The Itanium 2 processor includes hardware support for the `brl` instruction, which encodes a 64-bit offset using two slots of an instruction bundle.

There are a total of ten valid values for *brtype*, the branch type completer: none, `cond`, `call`, `ret`, `ia`, `cloop`, `ctop`, `cexit`, `wtop`, and `wexit`. Note that none is synonymous with `cond`. We defer discussions of some of the remaining completers until we describe the programming constructs that utilize them, while discussions of other branch type completers are omitted altogether; for these, we recommend the Itanium architecture documentation.

There are four valid values for *bwh*, the branch whether completer: `spnt`, `sptk`, `dpnt`, and `dptk`. The programmer or compiler can statically (`s`) predict (`p`) whether a branch will be taken (`tk`) or not taken (`nt`); prediction can also be completed dynamically (`d`) by the hardware. Branch prediction is an implementation-dependent feature, so the `sptk` hint is always available.

There are three valid values for *ph*, the prefetch hint completer: none, `few`, and `many`. Note that none is synonymous with `few`. This hint indicates how many lines should be prefetched into the Itanium instruction cache, beginning with the target instruction bundle.

There are two choices for *dh*, the branch cache deallocation hint completer: none and `clr`. This hints indicates whether the small cache dedicated to branch target addresses should be left alone (none) or flushed (`clr`). Note that the existence of such a cache is an implementation-dependent feature.

All branches incur time penalties. The Itanium architecture takes great pains to reduce the impact of branch instructions and includes many features that improve branch performance. For instance, the execution of a predicated branch instruction in a B-unit can be overlapped with the immediately prior comparison instruction that is computing the branch's predicate result in an I- or M-unit. The impact of the time required to compute the branch target's address is thus reduced by executing these instructions in parallel. This ability allows *zero latency* between the compare and branch instructions.

When combined with predication, the branching instructions give the programmer a powerful tool for controlling the logical flow of their programs.

7 Logical and Bit-Level Instructions

As we have seen, most Itanium instructions operate on the full 64-bit data value contained in a register. However, several instructions that can be used to manipulate individual bits within a register are also available; we discuss these instructions here.

7.1 Logical Instructions

Several Itanium logical instructions are available:

```
and      r1=r2,r3        // r1 <- r2 & r3
and      r1=imm8,r3      // r1 <- sext(imm8) & r3
andcm    r1=r2,r3        // r1 <- r2 &  r3
andcm    r1=imm8,r3      // r1 <- sext(imm8) &  r3
or       r1=r2,r3        // r1 <- r2 | r3
or       r1=imm8,r3      // r1 <- sext(imm8) | r3
xor      r1=r2,r3        // r1 <- r2 ^ r3
xor      r1=imm8,r3      // r1 <- sext(imm8) ^ r3
```

where `&`, `|`, and `^` denote the Boolean AND, OR, and XOR operations.

The logical functions supported by the Itanium ISA can be used to set, clear, toggle, and test individual bits within a value using bit-masking techniques. Evans and Trimper discuss the details of using the Itanium logical instructions to accomplish these common programming tasks (Section 6.1.3, page 158).

7.2 Bit-Level Instructions

We have already seen the `shladd` arithmetic instruction, which combines bit-shifting with integer addition. The Itanium ISA includes several other useful bit manipulation instructions.

7.2.1 Bit-Shift Instructions

Several forms of the Itanium bit-shift instructions are available:

```
shl      r1=r3,r2         // r1 <- r3 shifted left r2 bits
shl      r1=r3,count6     // r1 <- r3 shifted left count6 bits
shr      r1=r3,r2         // r1 <- r3 shifted right r2 bits
shr      r1=r3,count6     // r1 <- r3 shifted right count6 bits
shr.u    r1=r3,r2         // r1 <- r3 shifted right r2 bits
                          // unsigned
shr.u    r1=r3,count6     // r1 <- r3 shifted right count6 bits
                          // unsigned
```

where `count6` is a 6-bit unsigned immediate that specifies the shift count. Note that left shifts with `shl` are always unsigned, while `shr` produces an arithmetic shift, unless the `.u` instruction completer is specified.

If an immediate value is used with these instructions, then `shl` is actually an assembler pseudo-op for a special case of the Itanium deposit instruction, while `shr` and `shr.u` are pseudo-ops for special cases of the Itanium extract instruction. These instructions are introduced momentarily.

7.2.2 Shift Right Pair Instruction

The Itanium ISA also includes a "long shift" instruction, which shifts a bit pattern that is twice the width of a single register:

```
shrp    r1=r2,r3,count6      // r1 = [r2:r3] shifted right count6 bits
```

This instruction treats the two 64-bit source registers `r2` and `r3` as a "single" 128-bit value and shifts the bits `count6` positions to the right. The rightmost 64 bits of the result are placed in the destination register `r1`.

The `shrp` instruction can be used to rotate the bit pattern if the same register is specified for both source operands.

7.2.3 Extract and Deposit Instructions

Two instructions that enable the programmer to read or write any number of bits within a register are available.

The Itanium extract instruction isolates some contiguous block of bits from the source register and places those bits, right-justified, into the destination register:

```
extr      r1=r3,pos6,len6      // signed form
extr.u    r1=r3,pos6,len6      // unsigned form
```

where `pos6` is a bit position (in the range <63:0>) and `len6` specifies the number of bits to extract. Bits <`pos6+len6-1`:`pos6`> from the source register `r3` are copied into the destination register `r1` as bits <`len6-1`:0>. If the `.u` instruction completer is used, the remaining bits in the destination are set to zero; otherwise, bit <`pos6+len6-1`> from the source is propagated as the sign bit.

The Itanium deposit instruction isolates a contiguous span of bits from the right-hand side of the source register and repositions that span anywhere within the destination register:

```
dep.z    r1=r2,pos6,len6          // zero form
dep.z    r1=imm8,pos6,len6        // zero form, with immediate
dep      r1=r2,r3,pos6,len4       // merge form
dep      r1=imm1,r3,pos6,len6     // merge form, with immediate
```

where `pos6` is a bit position (in the range <63:0>), `len4` and `len6` specify the number of bits to deposit, and `imm1` and `imm8` are 1- and 8-bit immediate values.

The zero form copies bits <`len6-1`:0> from the source register `r2` into the destination register `r1`, setting all other bits of the destination to zero. If an immediate value is used, it is first sign-extended before bits <`len6-1`:0> are deposited into the destination.

The merge form copies bits <`len6-1`:0> from the source register `r2` into the destination register `r1` as bits <`post+len6-1`:`pos6`>, and all other bits of the destination are copied from the corresponding positions within the source register `r3`. Note that at most 16 bits can be copied from register `r2`, a result of the 4-bit width of `len4`. If an immediate value is used, it is sign-extended and bits <`len6-1`:0> serve as the first source segment.

Figure 2 (next page) demonstrates how these instructions operate, using quad word operands. In this figure, `pos6`=32 and `len6`=16.

Figure 2: The Itanium extract and deposit instructions

7.2.4 Single-Bit Test Instruction

The Itanium ISA also provides the `tbit` instruction to set qualifying predicates based on a test of any single bit within a 64-bit register:

```
tbit.trel.ctype    pt,pf=r3,pos6
```

where `pt` and `pf` are predicate registers and `pos6` is an unsigned value that encodes the particular bit position (in the range <63:0>) within the register `r3` that will be tested.

There are two valid values for *`trel`*, the test relationship completer: `nz` (nonzero) and `z` (zero). The predicate register `pt` is set to true and `pf` to false depending on the specified *`trel`* completer.

The valid values for the *`ctype`* completer are the same as those for the other comparison operations, which were discussed in Section 6.

8 Floating-Point Instructions

Floating-point data types and the now standard IEEE representations were introduced in Section 2, while the Itanium floating-point registers were introduced in Section 3. We turn our attention to the Itanium instructions that operate on the floating-point data types using the associated registers.

Many of the Itanium integer instructions have direct floating-point counterparts, while others do not. Those that do are listed in Table 9 with their floating-point analogues. Those integer instructions that do not have similar floating-point versions include the bit-shift instructions and data-independent branching instructions. Similarly, many floating-point instructions that we will discuss have no corresponding integer version.

Type of Instruction	Integer	Floating-Point
Arithmetic	`add`	`fadd (pseudo-op)`
	`sub`	`fsub (pseudo-op)`
	`xmpy (pseudo-op)`	`fmpy (pseudo-op)`
Load & Store	`ld1, ld2, ld4, ld8`	`ldfs, ldfd, ldfe, ldf8`
	`st1, st2, st4, st8`	`stfs, stfd, stfe, stf8`
Compare	`cmp`	`fcmp`
Logical	`and`	`fand`
	`andcm`	`fandcm`
	`or`	`for`
	`xor`	`fxor`

Table 9: Comparison of the Itanium integer and floating-point instructions

You will recall from our earlier discussions that the Itanium's 128 floating-point registers are 82 bits wide. This seemingly strange 82-bit width was chosen to accommodate Intel's 80-bit extended double-precision format, which has been carried over from the IA-32 architecture. As a consequence, these registers enable greater accuracy when using intermediate, "register-only" results. Floating-point values are converted to the appropriate IEEE format only when storing the final results of a calculation in memory.

In addition to the single- and double-precision floating-point formats, the IEEE standard defines encodings for various "special" values, like positive and negative infinity, for example. Table 10 lists these encodings and their meanings. Note that *denormal* numbers are those whose fractions have not been shifted far enough to the left to give the significand the "hidden bit" mentioned in Section 2. These numbers have a value between zero and the smallest of the normalized numbers given in Table 2 (on page 4).

Biased Exponent	Fraction	Meaning
All ones	Nonzero	NaN
All ones	Zero	$\pm$ Infi nity
Zero	Nonzero	$\pm$ Denormal
Zero	Zero	$\pm$ Zero
Other	Any	Nonzero, normalized

Table 10: Meanings of the special IEEE floating-point representations

8.1 Arithmetic Instructions

The Itanium architecture, like most modern designs, includes native support for some common floating-point arithmetic operations.

8.1.1 Addition, Subtraction, and Multiplication

The Itanium ISA provides for addition, subtraction, and multiplication for floating-point data types:

```
fadd.pc.sf     f1=f3,f2      // f1 <- f3 + f2
fsub.pc.sf     f1=f3,f2      // f1 <- f3 - f2
fmpy.pc.sf     f1=f3,f4      // f1 <- f3*f4
fnmpy.pc.sf    f1=f3,f4      // f1 <- -(f3*f4)
```

where `f1`, `f2`, `f3`, and `f4` can be any of the Itanium floating-point registers (Fr_0–Fr_{127}). However, unlike their integer counterparts, these floating-point instructions do not support immediate constants of any sort.

There are three valid values for *pc*, the precision completer: none, `s`, and `d`. None, which corresponds to omitting the precision completer, is used to handle special circumstances like the IA-32 double extended format. Our focus will be on the IEEE single- (`s`) and double- (`d`) precision.

Five valid values for *sf*, the status field completer, are available: none, `s0`, `s1`, `s2`, and `s3`. None is synonymous with `s0`. These values refer to four settings in a floating-point status register that we do not describe further. The default value (none) is used throughout the remainder of this section. Evans and Trimper describe the status field completer briefly (Section 8.4.5 on page 242). For the details concerning the floating-point status registers, consult the Itanium architecture documentation.

Each of the basic arithmetic operations described here is actually an assembler pseudo-op for special cases of the more general "fused" floating-point arithmetic instructions. We introduce these powerful instructions next.

8.1.2 Fused Multiply-Add and Multiply-Subtract

Three instructions that multiply two source operands and then add or subtract a third operand are also available:

```
fma.pc.sf     f1=f3,f4,f2      // f1 <- f3*f4 + f2
fms.pc.sf     f1=f3,f4,f2      // f1 <- f3*f4 - f2
fnma.pc.sf    f1=f3,f4,f2      // f1 <- -(f3*f4) + f2
```

where `f1`, `f2`, `f3`, and `f4` can again be any of the Itanium floating-point registers (Fr_0–Fr_{127}). Note that the intermediate product of `f3` and `f4` is not rounded in any way before adding or subtracting `f2`, producing a final value that is to the optimal precision.

The valid values for *pc* and *sf* are the same as those for the non-fused assembler pseudo-ops described above.

8.1.3 Reciprocal and Square Root Approximations

The IEEE standard also includes requirements for division, remainder, and unary square root operations with floating-point data types. Some RISC architectures provide full hardware support for these operations, but because they take longer to execute than other instructions, these operations tend to cause pipeline stalls.

In contrast, the Itanium architecture employs lookup tables that store approximations to reciprocals and reciprocal square roots with a known accuracy. The execution time for a table lookup is the same as for the `fma` instruction. The approximation obtained by these instructions can be refined to the desired accuracy using series expansion. As a result of instruction-level parallelism, this approach can be as fast as, if not faster than, the hardware-only implementations.

Floating-Point Reciprocal Approximation. Using two source operands and two destination operands, the reciprocal approximation instruction can compute either an approximate reciprocal or an IEEE-mandated quotient:

```
frcpa.sf    f1,p2=f2,f3       // p2 = 1 and f1 = 1/f3 or
                              // p2 = 0 and f1 <- f2/f3
```

where `sf` can take on any of the valid values from the previously discussed floating-point instructions. If the instruction is used with a qualifying predicate and that predicate is zero, then the predicate register `p2` is set to zero and the contents of `f1` remain unmodified. If the qualifying predicate is one, then either:

- `p2` is set to one and `f1` is set to the reciprocal of `f3`, or
- `p2` is set to zero and `f1` is set to the IEEE-mandated quotient, `f2/f3`.

Evans and Trimper include a brief discussion demonstrating how this instruction can be used to compute a refined result (Section 8.8.1, page 254). Their example follows that given by Peter Markstein in his book *IA-64 and Elementary Functions: Speed and Precision*. We recommend both of these resources for more details concerning the use of the `frcpa` instruction.

Floating-Point Reciprocal Square Root Approximation. In the same manner, the `frsqrta` instruction computes either an approximate reciprocal square root or an IEEE-mandated square root:

```
frsqrta.sf    f1,p2=f2,f3       // p2 = 1 and f1 = 1/sqrt(f3) or
                                // p2 = 0 and f1 <- sqrt(f3)
```

where the valid values for `sf` are as before. If the instruction is used with a qualifying predicate and that predicate is zero, then the predicate register `p2` is set to zero and the contents of `f1` remain unmodified. If the qualifying predicate is one, then either:

- `p2` is set to one and `f1` is set to the reciprocal of `sqrt(f3)`, or
- `p2` is set to zero and `f1` is set to the IEEE-mandated square root, `sqrt(f3)`.

Here, too, Evans and Trimper include a brief discussion demonstrating how this instruction can be used to compute a refined result (Section 8.8.2, page 255), again following Markstein. For the details, please consult either of these excellent resources.

Floating-Point Division. As noted, the Itanium architecture does not include an instruction for floating-point division, but virtually all programming environments provide some form of a software substitute. Consult your system's documentation to see which (possibly unpublished) internal routine or inline instruction sequence is used.

Markstein discusses numerous algorithms that compute these and various other mathematical functions using the Itanium instruction set while conforming to the IEEE conventions for rounding and exception reporting. Refer to his book for the details of using the Itanium floating-point instructions to compute refined arithmetic results.

8.1.4 Maximum and Minimum Instructions

The Itanium ISA includes instructions for determining the maximum or minimum of two floating-point values:

```
famax.sf    f1=f2,f3      // f1 <- larger of f2 and f3 (absolute value)
famin.sf    f1=f2,f3      // f1 <- smaller of f2 and f3 (absolute value)
fmax.sf     f1=f2,f3      // f1 <- larger of f2 and f3
fmin.sf     f1=f2,f3      // f1 <- smaller of f2 and f3
```

where the valid values for `sf` are as before. If the values in registers `f2` and `f3` are equal in value (`fmax`, `fmin`) or magnitude (`famax`, `famin`), then register `f1` is set to the value of `f3`.

8.1.5 Normalization

The Itanium ISA includes an assembler pseudo-op for "normalizing" and rounding a floating-point value after a series of calculations:

```
fnorm.pc.sf    f1=f3       becomes      fma.pc.sf    f1=f3,f1,f0
```

where the valid values for `pc` and `sf` are as before.

8.2 Data Access Instructions

Like their integer analogues, the floating-point load and store operations can use instruction completers to provide hints to the Itanium cache structures regarding how the program expects the values to be used. We discuss the available floating-point load and store instructions below.

8.2.1 Load Instructions

The Itanium ISA includes instructions for loading floating-point values in several different forms.

Standard floating-point load. There are three forms of the standard floating-point load instructions:

```
ldffsz.fldtype.ldhint    f1=[r3]          // f1 <- mem[r3]
ldffsz.fldtype.ldhint    f1=[r3],r2       // f1 <- mem[r3]
                                          // r3 <- r3 + r2
ldffsz.fldtype.ldhint    f1=[r3],imm9     // f1 <- mem[r3]
                                          // r3 <- r3 + sext(imm9)

ldf8.fldtype.ldhint      f1=[r3]          // f1<63:0> <- mem[r3]
ldf8.fldtype.ldhint      f1=[r3],r2       // f1<63:0> <- mem[r3]
                                          // r3 <- r3 + r2
ldf8.fldtype.ldhint      f1=[r3],imm9     // f1<63:0> <- mem[r3]
                                          // r3 <- r3 + sext(imm9)

ldf.fill.ldhint          f1=[r3]          // f1 <- mem[r3]
ldf.fill.ldhint          f1=[r3],r2       // f1 <- mem[r3]
                                          // r3 <- r3 + r2
ldf.fill.ldhint          f1=[r3],imm9     // f1 <- mem[r3]
                                          // r3 <- r3 + sext(imm9)
```

where *fsz* is the size of the information unit at the address specified in register `r3` from which a converted value is placed into register `f1`. The valid values for *fsz* are `s` for a single-precision value, `d` for a double-precision value, and `e` for an IA-32 80-bit extended double-precision value. Note that the load instruction uses register indirect addressing for the source operand and register direct addressing for the destination.

There are several valid values for *fldtype*, the load type completer. None, which corresponds to omitting the load type completer, indicates an ordinary load operation. The remaining types correspond to a check load, speculative load, or advanced load, and are considered further in Section 12.

Three valid values for *ldhint*, the load hint completer, exist: none, `nt1`, and `nta`. These completers provide the same hints to the Itanium cache structures as their integer counterparts, which were discussed in Section 5.

Also like the integer versions, the floating-point load instructions provide for postmodification of the pointer value in register `r3` by a full 64-bit signed value stored in register `r2` or by a 9-bit signed constant, with values ranging from -256 to +255.

The `ldf8` form of this instruction loads 8 bytes from the quad word memory location specified in register `r3` into the significand bits (<63:0>) of register `f1`. This form is typically used to load integer data types into a floating-point register. Accordingly, the sign bit (<81>) is set to zero and the biased exponent field (bits <80:64>) is set to 0x1003E (2^{63}) to indicate that the value in the significand bits should be interpreted as a 64-bit integer.

Finally, the `fill` form of this instruction loads 16 bytes and the appropriate fields are placed into register `f1` without conversion. This form is used to restore register contents when an operating system switches process contexts or when an application uses a preserved register.

Floating-point load pair. The Itanium ISA includes an instruction that will load a pair of floating-point values. Several forms are available:

```
ldfps.fldtype.ldhint   f1,f2=[r3]        // f1 <- mem[r3]
                                         // f2 <- mem[r3+4]
ldfps.fldtype.ldhint   f1,f2=[r3],8      // f1 <- mem[r3]
                                         // f2 <- mem[r3+4]
                                         // r3 <- r3 + 8

ldfpd.fldtype.ldhint   f1,f2=[r3]        // f1 <- mem[r3]
                                         // f2 <- mem[r3+8]
ldfpd.fldtype.ldhint   f1,f2=[r3],16     // f1 <- mem[r3]
                                         // f2 <- mem[r3+8]
                                         // r3 <- r3 + 16

ldfp8.fldtype.ldhint   f1,f2=[r3]        // f1<63:0> <- mem[r3]
                                         // f2<63:0> <- mem[r3+8]
ldfp8.fldtype.ldhint   f1,f2=[r3],16     // f1<63:0> <- mem[r3]
                                         // f2<63:0> <- mem[r3+8]
                                         // r3 <- r3 + 16
```

Data from two successive information units (starting at the address specified in register `r3`) are converted and placed into the destination registers `f1` and `f2`. Note that one of the destination registers must be an odd-numbered register and the other even-numbered, but they do not have to be consecutively numbered. (For example, specifying `f9` and `f12` as the destination registers is valid.) Other restrictions apply; consult the the Itanium architecture documentation for the details.

Like the standard floating-point load instruction, this operation provides for postmodification of the pointer value in register `r3` by an amount equal to the aggregate size of the two values, and is useful for stepping through an array, for example.

The valid values for *fldtype* and *ldhint* are the same as those for the standard floating-point load instructions.

8.2.2 Store Instructions

The Itanium ISA provides three forms of the floating-point store instruction:

```
stffsz.sthint       [r3]=f2            // mem[r3] <- f2
stffsz.sthint       [r3]=f2,imm9       // mem[r3] <- f2
                                       // r3 <- r3 + sext(imm9)

stf8.sthint         [r3]=f2            // mem[r3] <- f2<63:0>
stf8.sthint         [r3]=f2,imm9       // mem[r3] <- f2<63:0>
                                       // r3 <- r3 + sext(imm9)

stf.spill.sthint    [r3]=f2            // mem[r3] <- f2
stf.spill.sthint    [r3]=f2,imm9       // mem[r3] <- f2
                                       // r3 <- r3 + sext(imm9)
```

where *fsz* is the size of the information unit at the address specified by register `r3` into which the value in register `f2` is converted and stored. The valid values for *fsz* are `s` for a single-precision value, `d` for a double-precision value, and `e` for an IA-32 80-bit extended double-precision value. Note that the store instruction uses register direct addressing for the source operand and register indirect addressing for the destination.

There are several valid values for *fldtype*, the load type completer. None, which corresponds to omitting the load type completer, indicates an ordinary load operation. The remaining types correspond to a check load, speculative load, or advanced load, and are considered further in Section 12.

Two valid values for *sthint*, the store hint completer, exist: none and `nta`. These completers provide the same hints to the Itanium cache structures as their integer counterparts, which were discussed in Section 5.

Also like the integer versions, the floating-point store instructions provide for postmodification of the pointer value in register `r3` by a 9-bit signed constant, with values ranging from -256 to +255.

The `stf8` form of this instruction stores the significand bits (<63:0>) of register `f2` in the quad word memory location specified by register `r3`.

Finally, the `spill` form of this instruction stores the contents of register `f2` into the 16-byte memory location specified by `r3`. This form is used to save register contents when an operating system switches process contexts or when an application uses a preserved register.

8.3 Miscellaneous Floating-Point Instructions

The Itanium ISA includes a few other instructions that operate on floating-point data types. We discuss these here.

8.3.1 Floating-Point Compare Instruction

In Section 6, the notion of predication was introduced. The Itanium ISA includes a floating-point compare instruction that can be used to set qualifying predicate registers based on the value in a floating-point register. The behavior and syntax of this instruction is similar to that of its integer analogue:

```
fcmp.fcrel.fctype    p1,p2=f2,f3      // always uses two registers
```

where two predicate registers `p1` and `p2` must always be specified and can be any of Pr_0–Pr_{63}.

Typically, a comparison statements is read from left to right: `p1` is set to true and `p2` to false if `r2 crel r3` is true, and vice versa if the comparison is false.

There are several valid values for *fcrel*, the conditional relationship completer, including: `eq`, `ne`, `lt`, `le`, `ge`, `gt`. Each of these has the same meaning as the corresponding symbol in the integer versions of the compare instruction. A number of other values for the conditional relationship completer are also available: `nlt`, `nle`, `nge`, and `ngt`. Here, `n` stands for "not", so these completers provide mnemonics for the logically opposite relationships.

The IEEE standard also defines a special "unordered" relation that is true if one or both operand values are NaN (not a number). The `fcmp` instruction can use the `unord` completer to test for this relationship and the `ord` completer for the Boolean opposite.

Two valid values for *`fctype`*, the comparison type, exist: none and `unc`. None, which corresponds to omitting the comparison type completer, indicates an ordinary comparison, like that just described. The `unc` completer indicates and unconditional comparison and behaves just like the unconditional integer comparison operation, which was described in Section 6.

8.3.2 Logical Instructions

The Itanium ISA includes logical instructions that operate on the significand of a floating-point value:

```
fand      f1=f2,f3       // significand of f1 <- f2 & f3
fandcm    f1=f2,f3       // significand of f1 <- f2 &  f3
for       f1=f2,f3       // significand of f1 <- f2 | f3
fxor      f1=f2,f3       // significand of f1 <- f2 ^ f3
fselect   f1=f3,f4,f2    // significand of f1 <- (f3&f2) ^ (f4&f2)
```

where `&`, `|`, and `^` denote the Boolean AND, OR, and XOR operations. Each instruction sets the sign of `f1` to positive and the biased exponent field to 0x1003E.

The `fselect` instruction copies significand bits of `f3` from the positions where the bits of `f2` are one and it copies the bits of `f4` from the positions where the bits of `f2` are zero.

8.3.3 Assembler Pseudo-Ops

The Itanium ISA also provides a number of assembler pseudo-ops for copying floating-point values between registers. These include:

```
mov       f1=f3     // f1 <- f3
fabs      f1=f3     // f1 <- abs(f3)
fneg      f1=f3     // f1 <- -f3
fnegabs   f1=f3     // f1 <- -abs(f3)
```

where `f1` and `f3` may be any of the Itanium floating-point registers (Fr_0–Fr_{127}). Like all assembler pseudo-ops, these represent common special cases of the more general floating-point instructions discussed above.

8.3.4 Floating-Point Merge Instruction

We know that floating-point numbers are stored and manipulated as sign and magnitude quantities. There are several forms of a merge instruction to manipulate the sign bit of a floating-point number, both with and without the biased exponent field:

```
fmerge.s    f1=f2,f3     // f1 <- sign(f2) with rest(f3)
fmerge.ns   f1=f2,f3     // f1 <- -sign(f2) with rest(f3)
fmerge.se   f1=f2,f3     // f1 <- sign(f2) and exp(f2) with rest(f3)
```

where `f1`, `f2`, and `f3` may any of the Itanium floating-point registers (Fr_0–Fr_{127}). These instructions are useful for composing a new floating-point value using a combination of the various elements from the source operands.

8.3.5 Floating-Point Value Classification

The `fclass` instruction allows the programmer to determine the (nature) of the current value in a floating-point register:

```
fclass.fcrel.fctype   p1,p2=f2,fclass9      // is f2 as expected?
```

where two predicate registers must always be specified and `fclass9` is a bit pattern encoding the characteristics sought about the contents of register `f2`.

Predicate register `p1` is set to true and `p2` to false if `f2` *`fcrel`* `fclass9` is true, and vice versa if the relationship is false.

There are two valid values for *`fcrel`*, the conditional relationship completer: `m` (is a member) and `nm` (is not a member).

There are two valid values for *`fctype`*, the comparison type completer: none and `unc`. None, which corresponds to omitting the completer, indicates an ordinary comparison. The `unc` completer indicates an unconditional comparison, and behaves as previously described.

Itanium assemblers will recognize the mnemonics in Table 11 for the `fclass9` bit pattern. These mnemonics can be OR'd together using the `|` operator.

Floating-Point Class	Mnemonic	Bit Value in fclass9
NaTVal	@nat	0x100
Quiet NaN	@qnan	0x080
Signaling NaN	@snan	0x040
Positive	@pos	0x001
Negative	@neg	0x002
Zero	@zero	0x004
Un-normalized	@unorm	0x008
Normalized	@norm	0x010
Infi nity	@inf	0x020

Table 11: Assembler mnemonics for the `fclass` instruction

The floating-point number will agree with the `fclass9` pattern if one of the following three conditions is true:

- The value is NaTVal and `@nat` was specified.
- The value is NaN and either `@qnan` or `@snan` was specified.
- The value's sign agrees with `@pos` or `@neg`, if specified, and the value's type agrees with the remainder of the specified characteristics.

Note that a value of 0x1ff for `fclass9` will test whether the value in register `f2` is any supported floating-point type.

8.4 Floating-Point Operations on Integer Values

We noted in Section 5 that the Itanium floating-point registers could be used to do full width multiplication of 64-bit integers. Before investigating the several forms of the floating-point instruction that makes this operation possible, we introduce the instructions for converting between integer and floating-point representations.

8.4.1 Data Conversion

We have already seen that the floating-point load and store operations will convert data to and from the IEEE single- and double-precision formats when working with data in the floating-point registers. The Itanium ISA also includes instructions for converting quad word signed integer values as well.

Rounding and truncation. Several instructions that modify the format of a floating-point value as it moves between Itanium registers are available:

```
fcvt.fx.sf           f1=f2      // round to integer
fcvt.fx.trunc.sf     f1=f2      // truncate to integer
fcvt.fxu.sf          f1=f2      // round to unsigned integer
fcvt.fxu.trunc.sf    f1=f2      // truncate to unsigned integer
```

where the result of each operation is placed into the significand of register `f1`. The biased exponent of `f1` is set to 0x1003E and the sign bit is set to zero. If the floating-point value in register `f2` is negative, then the sign of the result in `f1` is given by bit <63> of the significand. This qualification applies only to the signed forms of these instructions.

The valid values for *sf* are the same as those previously described for the floating-point arithmetic instructions.

Integer to floating-point conversion. The Itanium ISA also provides an instruction for converting a 64-bit integer (stored in the significand of a floating-point register) into a normalized floating-point value:

```
fcvt.xf     f1=f2      // convert to normalized floating-point
```

This operation is always exact, a result of the extended exponent range of the Itanium floating-point registers. Note that no instruction completers are necessary. An assembler pseudo-op that converts a 64-bit integer into a floating-point value using the `fma` instruction is also available:

```
fcvt.xuf.pc.sf   f1=f2       becomes      fma    f1=f3,f1,f0
```

Rounding may be necessary if the integer value is too large; a truncation operation is not available.

The valid values for *pc* and *sf* are the same as those for the floating-point arithmetic operations.

Data movement. Several forms of the `getf` instruction, which moves values from the floating-point registers to the general-purpose registers, are available:

```
getf.s      r1=f2      // r1 <- single-precision representation of f2
getf.d      r1=f2      // r1 <- double-precision representation of f2
getf.exp    r1=f2      // r1<17:0> <- sign and exponent of f2
getf.sig    r1=f2      // r1 <- significand of f2
```

Bits <63:32> of `r1` are set to zero with `getf.s`. Likewise, bits <63:18> of `r1` are set to zero with `getf.exp`. If `f2` contains NaTVal, then the NaT bit of `r1` is marked for all forms of this instruction.

Similar instructions are available for moving values from a general-purpose register to a floating-point register:

```
setf.s      f1=r2      // f1 <- single-precision value in r2
setf.d      f1=r2      // f1 <- double-precision value in r2
setf.exp    f1=r2      // sign and exponent of f1 <- r2<17:0>
setf.sig    f1=r2      // significand of f2 <- r2
```

Here, bits <17:0> of `r2` are set in the sign and exponent fields of `f1`, and its significand is set to the hexadecimal value 1 followed by 15 zeros (0x1000000000000000) with the `setf.exp` instruction. With `setf.sig`, the value in `r2` is copied into the significand of `f1`, its sign filed is set to zero, and the biased exponent is set to 0x1003E. If the NaT bit of register `r2` is set, then the conversion is skipped and register `f1` is set to NaTVal.

8.4.2 Integer Multiplication

The Itanium ISA provides a fused multiply-add instruction for multiply 64-bit integer data types stored in floating-point registers. Several forms of the `xma` instruction are available (some of which are assembler pseudo-ops):

```
xma.l      f1=f3,f4,f2      // low form
xma.lu     f1=f3,f4,f2      // unsigned low form
xma.h      f1=f3,f4,f2      // high form
xma.hu     f1=f3,f4,f2      // unsigned high form
xmpy.l     f1=f3,f4         // low form (pseudo-op)
xmpy.lu    f1=f3,f4         // unsigned low form (pseudo-op)
xmpy.h     f1=f3,f4         // high form (pseudo-op)
xmpy.hu    f1=f3,f4         // unsigned high form (pseudo-op)
```

where a 128-bit intermediate result is formed by either a signed or unsigned multiplication of the significands of registers `f3` and `f4` and (possibly) adding the significand of register `f2`. Note that the significand of `f2` is zero-extended as necessary. Either the lower or the upper 64 bits of this results are then stored in significand of register `f1`.

There is no fused multiply-subtract instruction for 64-bit integer data types, a result of zero-extending, and not sign-extending, the significand of `f2`.

The sign bit of `f1` is set to zero, and the biased exponent to 0x1003E. If any source operand's value is NaTVal, then the conversion is skipped and `f1` is set to NaTVal.

9 Parallel Instructions

As we know, the Itanium architecture operates on 64-bit integer and 82-bit floating-point data types. Sometimes, however, the precision or range of values enabled by the full width of the appropriate registers are not necessary. To optimize the circumstances where this situation is true, many modern architectures offer parallel (sometimes called "multimedia") instructions. These instructions operate on several narrow data types, packed into a full-width architectural register, in parallel.

The Itanium architecture provides several integer and floating-point parallel instructions. The use of parallel instructions is extremely complex and often requires special algorithms and data layouts to achieve optimal execution. As a result, we give only a cursory overview of these instructions here.

For further details concerning the Itanium parallel instructions, consult the appropriate entries in Section 3 of the Itanium Instruction Set Reference.

9.1 Integer Instructions

A large number of parallel integer instructions, which perform their specified operations on multiple bytes, words, and double words packed into a 64-bit general-purpose register, are available, but we do not list the instruction mnemonics here.

The parallel integer instructions include: typical arithmetic operations, like add, subtract, multiply; many useful others, such as maximum, minimum, average, bit shifts, comparisons, etc.; and the necessary instructions for packing and unpacking general-purpose registers with multiple narrow data types.

Note that these instructions have latencies greater than one and may require execution unit I0 in particular. There use is further complicated by interdependencies with other Itanium instructions. It is possible that the analogous nonparallel instructions will exhibit better performance, but this result is, of course, implementation-dependent.

9.2 Floating-Point Instructions

Similarly, many parallel floating-point instructions, which perform their specified operations on two single-precision floating-point values packed into an 82-bit Itanium floating-point register, are available, but we do not list the instruction mnemonics here.

The parallel floating-point instructions include: typical arithmetic operations, like the fused multiply-add and multiply-subtract; many useful others, such as maximum, minimum, negation, comparison, the reciprocal approximations, etc.; and the necessary instructions for loading and storing the packed data values.

These instructions have a four-cycle latency. In principle, then, the Itanium architecture can sustain twice as many parallel single-precision operations as nonparallel double-precision operations. Of course, data dependencies and other factors, like the number of F-units, will limit the achievable speed-up provided by the parallel instructions.

10 Structured Programming Constructs

We saw in Section 6 that the Itanium ISA includes several comparison and branching instructions to control the flow of a program's execution. Many of the common control structures provided by high-level languages, including logical or data-dependent constructs (`if...then...else` and case selection structures) and loops (`do...until` or `while...do`), can benefit from implementation using the Itanium's advanced features. We turn our attention to the low-level bases for building these high-level constructs using the Itanium ISA.

10.1 If...Then...Else Structures

The `if...then...else` block is one of the simplest structured programming constructs. It is also one of the most powerful.

10.1.1 Standard Implementation

An assembly level `if...then...else` construct will typically look like the following:

```
        <prior code>
if:     cmp.crel pt,pf=ra,rb    // predicates for a rel b
        (pf) br.cond else;;     // skip THEN block
then:   <do THEN block>;;
        br end;;                // skip ELSE block
else:   <do ELSE block>;;
end:    <subsequent code>
```

where `pt` and `pf` are the predicate registers set for the true and false outcomes of the compare instruction. When the outcome is true, `pf` will contain a zero, the first conditional branch will fall through, the `THEN` block will execute, and the unconditional branch will skip the `ELSE` block. When the outcome is false, `pf` will contain a one, the first conditional branch skips the `THEN` block, and the `ELSE` block will execute. Note that in either case one branch must execute, which is very time-consuming.

10.1.2 Predicated Implementation

Most Itanium instructions can be predicated, so both branch instructions encountered above can be eliminated:

```
        <prior code>
if:     cmp.crel pt,pf=ra,rb;;  // predicates for a rel b
then:   (pt) <do THEN block>
else:   (pf) <do ELSE block>
end:    <subsequent code>
```

Predication of the `THEN` and `ELSE` blocks can impact performance significantly. The CPU executes both streams of instructions, but the values in the predicate registers determine which stream actually has an effect when results written into destination registers or memory.

Instructions that are to execute regardless of the construct's comparison outcome can be interleaved at the desired position within the instruction sequence. Most other architectures would require that these instructions be duplicated in each code block.

You will recall that the Itanium architecture enables zero latency between compare and branch instructions by performing the operations in separate I- or M-units (the compare) and B-units (the branch). Without an explicit stop (`;;`) after the compare instruction, the CPU is permitted to execute the compare and the predicated instructions of the `if...then...else` construct in parallel, possibly using stale values in

the predicate registers. The explicit stop ensures that the comparison has completed before the qualifying predicates are used.

When there are only a few instruction in the `THEN` and `ELSE` blocks, the programmer or compiler should remove as many stops as possible. Also, interleaving the small number of instructions from the `THEN` block with those from the `ELSE` block can ensure that instruction bundles are filled with useful work rather than no-ops.

Because the CPU actually executes the instructions from both code blocks and uses predication to determine which instructions have an effect, lengthy `THEN` and `ELSE` blocks may negatively impact the CPU's overall throughput. Although branch instructions are expensive, there is a performance trade-off between the predicated and standard implementations of the `if...then...else` construct, and at some point the standard version will execute more quickly. Where this crossover point lies is, of course, an implementation-dependent result.

Similarly, when the number of instructions in each block is severely imbalanced, using the standard `if...then...else` implementation may be more effective. This situation is particularly true if the shorter block belongs to the more probable outcome, because the CPU is executing a large number of instructions that have no effect (those that are predicated false) only to ensure the effect of a small number of instructions (those that are predicated true). With the standard implementation, the execution time will reflect only those few instructions and the one branch taken.

10.1.3 Nested If...Then...Else Structures Using Predication

You learned earlier that the unconditional comparison instruction, when predicated false, sets the values in both predicate registers to false without actually performing a comparison. This form of the comparison instruction is useful for implementing nested `if...then...else` structures, as follows:

```
        <prior code>
if:     cmp.crel pt,pf=ru,rv              // outer conditional test
then:   (pt) cmp.crel.unc pa,pb=rw,rx     // THEN-block conditional test
        (pa) <do inner THEN block A>
        (pb) <do inner ELSE block B>
else:   (pf) cmp.crel pc,pd=ry,rz         // ELSE-block conditional test
        (pc) <do inner THEN block C>
        (pd) <do inner ELSE block D>
end:    <subsequent code>
```

This branch-free implementation can be very useful for short code blocks. As before, all code blocks are loaded and executed, but only the sequence that has been predicated true will have an effect. If one or more of the code blocks is very long, consider using a branching implementation.

10.2 Case Selection Structures

Case selection structures can be implemented very compactly using the Itanium compare instructions and predication. Consider the following simple example, expressed in the C programming language:

```
switch ( W )
{
      case 1:
            R = P - Q;
            break;
      case 2:
            R = P + Q;
            break;
      case 4:
            R = P;
            break;
}
```

Using a sequence of compare instructions with predication, this code becomes:

```
          cmp.eq p1,p0=1,rW    // if rW == 1, p1 <- 1
          cmp.eq p2,p0=2,rW    // if rW == 2, p2 <- 1
          cmp.eq p4,p0=3,rW;;  // if rW == 4, p4 <- 1
case1:    (p1) sub rR=rP,rQ    // R = P - Q
case2:    (p2) add rR=rP,rQ    // R = P + Q
case4:    (p4) mov rR=rP;;     // R = P
```

where the notation `rR` means the general-purpose register containing the value of `R`, and so forth. The Itanium architecture allows more than one instruction to target the same register (in this example, the register `rR`) if those instructions are mutually exclusive; that is, if only one of the instructions will be predicated true. If this were not the case, then more stops would be required.

10.3 Loop Structures

Like the `if...then...else` construct and case selection structures, various types of loop structures can be implemented using predicated instruction execution.

10.3.1 Counter-controlled Loops

A counter-controlled loop can be expressed as follows:

```
        <enter loop with rc = number of traversals>
loop:   <instructions of loop body>
        add     rc=-1,rc;;              // decrement loop counter
        cmp.eq  p0,pf=rc,0              // is loop counter == 0?
        (pf) br.cond.sptk loop;;        // no, execute loop again
        <subsequent code>               // yes, continue
```

This loop uses only one predicate register. Furthermore, the static prediction hint (`sptk`) is given, so this loop assumes many traversals.

10.3.2 Loops Controlled by an Address Limit

An address can also be used to control the execution of a loop. For example, consider the following code that processes the quad word elements of an array:

```
        <enter loop with rc = address of first element>
loop:   ld8      rt=[rc],8;;             // load current element and
                                         // increment pointer
        <process current element>
        cmp.gtu  p0,pf=rc,rl             // past the last element?
        (pf) br.cond.sptk loop;;         // no, execute loop again
        <subsequent code>                // yes, continue
```

where `rl` contains the address of the last element. Note that an unsigned comparison is used for the addresses. Also, a loop of this sort implicitly assumes that the array is non-null.

10.3.3 Loops with a Conditional Entrance

Suppose that it is permissible for the array in the previous example to be null. Clearly this situation calls for a different sort of loop, one that would not attempt to operate on an empty array. We can remedy the situation by positioning the conditional test at top of the loop:

```
        <enter loop with rc = address of first element>
loop:   cmp.gtu  pt,p0=rc,rl             // past the last element?
        (pt) br.cond.spnt leave;;        // yes, exit the loop
        ld8      rt=[rc],8;;             // load current element and
                                         // increment pointer
        <process current element>
        br       loop;;                  // look for next element
leave:  <subsequent code>
```

where the notational conventions are as before. Here, we provide the static prediction hint `spnt`, indicating that the branch will not be taken.

10.3.4 Using the Loop Count Register

Counter-controlled loops are very common structures in application programming. Often, these structures are nested, so the need to handle the innermost loops efficiently is very important.

The Itanium architecture provides two mechanisms to implement loops efficiently: the `ar.lc` (loop count) application register and the `br.cloop` branch instruction.

The `ar.lc` register must be initialized, prior to entering the loop body, to one less than the total number of desired traversals using a `mov` pseudo-op. The `br.cloop` instruction, placed at the bottom of the loop, tests the value of `ar.lc` against zero after each traversal. If `ar.lc` is not zero, it is decremented and the branch is taken. If `ar.lc` is zero, the branch is not taken and execution falls through to the next instruction.

Here, the body of the loop will be executed at least once. If this is not the desired behavior, programmers or compilers must add the appropriate tests prior to the beginning of the loop body.

We have included another (slightly modified) example from Evans and Trimper to illustrate how `ar.lc` and `br.cloop` are used. The program, called `DOTCLOOP`, computes the scalar product of two vectors. The `DOTCLOOP` code is given in Figure 3 (next page).

The Itanium architecture provides only one `ar.lc` register, so any routine that uses the register must save and restore its contents for the calling routine. In this example, the general-purpose register `r9` can be used because the program's `main` procedure is a "leaf" procedure; that is, `main` does not call any other routines. Generally, however, a stack-based mechanism should be used to save and restore register contents for previous calling levels.

```
// DOTCLOOP: Compute the scalar product of two vectors

        N        = 3              // Declare a constant
        .data                     // Declare data section
        .align   8                // Specify desired alignment
P:      .skip    8                // To store the product
X:      data2    -1,+3,+5         // First vector of 16-bit values
Y:      data2    -2,-4,+6         // Second vector of 16-bit values

        .text                     // Declare code section
        .align   32               // Specify desired alignment
        .global  main             // Mark mandatory 'main' program entry
        .proc    main
main:
        .prologue                 // Begin prologue section
        .save    ar.lc,r9
        mov      r9=ar.lc;;       // Save caller's ar.lc
        .body                     // Begin procedure 'main'
first:  movl     r14=X;;          // Gr14 = pointer to X
        movl     r15=Y;;          // Gr15 = pointer to Y
        movl     r16=P;;          // Gr16 = pointer to P
        mov      r20=0            // r20 = scalar product
        mov      r17=N-1;;        // One less than the number of traversals
        mov      ar.lc=r17        // Initialize ar.lc

top:    ld2      r21=[r14],2      // Load element from X and increment pointer
        ld2      r22=[r15],2;;    // Load element from Y and increment pointer
        pmpy2.r  r21=r22,r21;;    // Multiply element from X by element from Y
        sxt4     r21=r21;;        // Sign-extend result to 64 bits
        add      r20=r20,r21      // Update scalar product
        br.cloop.sptk.few top     // More elements to process?
        st8      [r16]=r20;;      // No, store the scalar product

done:   mov      r8=0;;           // Signal completion
        mov      ar.lc=r9         // Restore caller's ar.lc
        br.ret.sptk.many b0;;     // Return to command line
        .endp    main             // End procedure `main`
```

Figure 3: A slightly modified version of the DOTCLOOP program from Evans and Trimper

In addition, the DOTCLOOP program introduces the prologue section of a program. The prologue, marked by .prologue, occurs at the beginning of the text segment and extends until the .body directive. You can see that the prologue includes both assembler directives (.save, for example) and actual Itanium instructions. Programs may require an epilogue section as well. Not surprisingly, the epilogue occurs at the bottom of the text segment, but there are no special directives to explicitly mark the epilogue section.

11 Using Procedures and Functions

We now turn out attention to the Itanium mechanisms, instructions, and calling conventions that support procedures and functions.

11.1 Itanium Stack Structures

The Itanium architecture includes support for both memory-based stacks and register stacks. We describe both types here, highlighting only those details necessary to provide a basic understanding of the low-level mechanisms prescribed by the Itanium architecture that support the use of procedures and functions.

11.1.1 Itanium Memory Stacks

By convention, the general-purpose register Gr_{12} serves as the Itanium *stack pointer*, and it is initialized to point to the memory stack when a program is loaded. The stack pointer requires 16-byte alignment, also by convention. Note that Itanium assemblers will recognize `sp` as a synonym for Gr_{12}.

Calling procedures will automatically provide a 16-byte "scratch" area for the callee. If more than 16 bytes are required, a *procedure frame* must be established: the called procedure must decrement `sp` by the *frame size* in its prologue section.

The procedure frame contains five areas: the local storage region, a dynamic allocation region, the frame marker region, and an outgoing parameters region. We omit any further details of the procedure frame, but Evans and Trimper cover the topic thoroughly (Section 7.1.3, page 191). We note that:

- The frame size must always be a multiple of 16 bytes.
- It is the responsibility of the programmer (or compiler) to save the previous stack pointer in the prologue and restore it in the epilogue.
- The Itanium architecture allows the programmer or compiler to define any number or type of stack structures in a program's data segment and to use the general-purpose registers as user-maintained stack pointers.

Evans and Trimper also cover more details of user-defined stacks (Section 7.1.4, page 192).

11.1.2 Itanium Register Stacks

In addition to memory stacks, the Itanium architecture supports a hardware-based register stack. The architecture prescribes that 32 static registers and a register stack of at least 96 registers (managed by the *register stack engine*, which we discuss momentarily) be provided by any implementation. We discuss this topic briefly, leaving the details to Evans and Trimper (Section 7.3, page 196) and other resources.

The alloc instruction. A new stack frame on the Itanium register stack is allocated using the following instruction:

```
alloc    r1=ar.pfs,ins,locs,outs,rots
```

where `ins`, `locs`, `outs`, and `rots` specify the sizes of the input, local, output, and rotating regions of the stack frame. The *size of the frame* (*sof*) is given by `ins + locs + outs`. The *size of the local region* (*sol*) is given by `ins + locs`; there is no distinction between the inputs and the locals. The *size of the rotating region* (*sor*) is given by `rots`, which must be a multiple of eight and cannot exceed *sof*

The `alloc` instruction has introduced the idea of *rotating registers*. This powerful concept allows data in registers to remain accessible by incrementing the logical names of the registers within the set using special instructions. We defer any further discussion of rotating register sets until Section 12.

The Register Stack Engine. The Itanium Register Stack Engine (RSE) provides transparent access to a large virtual register stack in memory. In this sense, it functions similarly to a typical cache structure. Using cues from the operating system, the RSE manages the limited number of physical registers in the stack by spilling and filling register contents to a *backing store* (typically a dedicated region of memory) when a new allocation or procedure return requires that action.

The RSE asynchronously moves data to and from memory without direct intervention by the CPU using direct memory access. In this sense, the RSE functions as an I/O device that is active only periodically and whose operation is largely decoupled from instruction execution. Of course, if the RSE is waiting for data from the CPU, or vice versa, execution may stall.

11.2 Calling Procedures and Functions

In order to reduce possible contention over the vast register resources provided by the Itanium architecture, programmers and compilers must follow the conventions for using these resources. Section 3 touched on the standardized uses of each type of Itanium register. Here, we describe the conventions for calling procedures and functions more fully.

11.2.1 Register Conventions

Many conventions reflect differences at the hardware level. For example, some registers are global in scope (Gr_0–Gr_{32}), others have constant values (Gr_0, Fr_0–Fr_1), and yet others are managed by the RSE. Following Evans and Trimper, we characterized the Itanium registers according to their size, features, and uses in Section 3. We reiterate the characterizations pertaining to the use of procedures and functions:

- A register is *scratch* if it may be used freely by a procedure or function at any calling level; the caller must save any important contents of these registers.
- A register is *preserved* if a calling routine depends on its contents; any called procedure must save and restore the contents of these registers for its caller.
- A register is *automatic* if its name only has a dynamic correspondence to a physical register; these registers are automatically spilled to and filled from memory during allocation by the hardware, as necessary.

Note that the location within the program where the contents of important registers are saved depends upon the programming language and environment, as well as the operating system.

11.2.2 Call and Return Branch Instructions

The Itanium architecture provides instruction completers for the more general branch instructions to indicate a call or return from a function:

```
br.call.bwh.ph.dh    b1=target25      // IP-relative
br.call.bwh.ph.dh    b1=b2            // indirect addressing
br.ret.bwh.ph.dh     b2               // indirect addressing only
```

where the valid values for *bwh*, *ph*, and *dh* are the same as those for the branch instructions described in Section 6. Note that the target address must be aligned with an instruction bundle; that is, the four lowest-order bits of the address must be zero. Also, note that calls and returns may have a qualifying predicate.

As a result of the `br.call` instruction, the return address becomes IP+16 and is stored in register `b1`. Then, several values are saved into the `ar.pfs` application register; we do not provide the details of this step, but they can be found in Appendix D.7 of Evans and Trimper. The register stack is adjusted and, finally, the IP is set to the target address either by adding the sign-extended offset, `target25`, or by copying the address in register `b2`.

The `br.ret` instruction copies the value of `b2` into the IP and restores from `ar.pfs` those values saved by `br.call`. The calling procedure's stack frame is also restored.

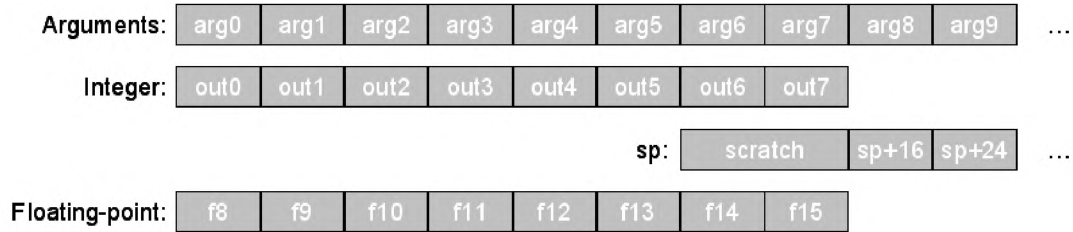


Figure 4: Passing arguments via registers and the memory stack

11.2.3 Argument Passing

The power of procedures and functions comes from their ability to take input arguments, operate on those arguments, and possibly return a result to the calling procedure.

Calling routines can pass as many as eight input arguments using registers; beyond eight, they must be passed using the memory stack. Up to eight general-purpose registers are therefore allocated as `outs` and are used to pass 64-bit arguments. Floating-point registers Fr_8–Fr_{15} are used to pass single- or double-precision floating-point arguments. Stack space for passing more than eight arguments can be claimed by decrementing the stack pointer by the appropriate amount (including the required 16-byte scratch space). Figure 4 shows schematically how argument passing works.

The rules for passing integer and floating-point data types differ. For instance, suppose we make the following C function call:

```
result = my_function( a, b, r, s, t, c, x, d, y );
```

where `a`, `b`, `c`, and `d` are integer values and `r`, `s`, `t`, `x`, and `y` are floating-point values. These arguments must correspond to the sequentially numbered argument slots, as depicted in Figure 4: `a` with `arg0`, `b` with `arg1`, and so forth.

Integer arguments in `arg0–arg7` are placed in the corresponding output registers `out0–out7`. Other output registers are not used. In contrast, floating-point arguments are placed in sequentially numbered floating-point registers `f8–f15`; registers are not skipped. Any remaining arguments (`arg8` and above) are stored in quad word information units beginning at `sp+16`, where `sp` is the decremented stack pointer that will be used by the callee.

So, for `my_function`, `a` is passed in `out0`, `b` in `out2`, `r` in `f8`, `s` in `f9`, `t` in `f10`, `c` in `out5`, `x` in `f11`, `d` in `out7`, and `y`, the ninth argument, is stored at `sp+16`. Note that `out3`, `out4` and `out6` are unused.

11.2.4 A Practical Example

To illustrate the preceding topics, we again include some (slightly modified) example code from Evans and Trimper. Two listings, `BOOTH` and `DECNUM3`, combine to form a program for converting a positive integer value into a string of ASCII encoded decimal digits. Figure 5 (page 48) shows the code for a function that computes a 128-bit signed product from two 64-bit inputs using Booth's algorithm, and Figure 6 (page 49) shows the code for the test program.

Rather than concentrate on the algorithms, we choose to highlight only those features of the code that are pertinent to the discussion at hand. The details of Booth's algorithm, as well as the process of converting integers in hexadecimal representation into ASCII encoded decimal digits, are covered by Evans and Trimper (Section 6.5.1, pages 170–173; Section 6.6, pages 175–178; Section 7.2, pages 194–196; Section 7.6, pages 214–218).

BOOTH. The `booth` function takes two 64-bit integers, multiplies them together, and produces a 128-bit signed product. The function expects the multiplicand to be in the first argument slot and the multiplier to be in the the second. To satisfy these requirements, the calling procedure must place the multiplicand on the register stack in `out0` and the multiplier in `out1`; these registers are accessible within the `booth` function as `in0` and `in1`. The function "claims" these two input registers using the `.regstk` directive:

```
.regstk    ins, locs, outs, rots
```

where `ins`, `locs`, and `outs` determine the *sof*, and `rots` specifies the number of rotating registers.

Typically, integer functions will return a value to their caller in register `r8` (`ret0`). The Itanium architecture permits up to four integer return values to be placed in general-purpose registers. Booth's algorithm computes a full, 128-bit product, which is clearly larger than a single general-purpose register. Thus, the function makes the low-order bits of the resulting product available to the caller by placing it in `ret0`; the high-order bits are placed in register `r9` (`ret1`).

Note that the `booth` function largely uses the scratch registers and does not allocate a new stack frame. The function does save the caller's `ar.lc` register (a preserved register) in `r31`.

DECNUM3. The `DECNUM3` test program also makes use of the scratch registers. However, because the contents of these registers is undefined upon the return from a procedure call (in this example, the call to `booth`), the calling routine is responsible for saving and restoring any important values of these registers. As a consequence, Evans and Trimper point out that, when designing and using procedure calls, it is important to enumerate the registers that must be preserved and then devise an efficient way to save and restore their contents.

The `DECNUM3` program allocates a new stack frame, using the `alloc` instruction, that will hold the six local and two output values. Table 12 lists the registers in the `DECNUM3` stack frame and their uses.

Register	Purpose
`loc0 (r32)`	Preserve `rp`
`loc1 (r33)`	Preserve `ar.pfs`
`loc2 (r34)`	Pointer to user-defi ned stack
`loc3 (r35)`	Approximation to 0.8
`loc4 (r36)`	Previous quotient for the remainder calculation
`loc5 (r37)`	Preserve `gp (r1)`
`out0 (r38)`	Pass multiplicand to `booth`
`out1 (r39)`	Pass multiplier to `booth`

Table 12: The registers, and their uses, of the DECNUM3 stack frame

We close this discussion by noting that the expected result is 4,888,718,345, the decimal representation of 0x123456789. The program can be tested by using the debugger included with your programming environment to inspect the contents of the 80 bytes starting at address `A3`.

```
// BOOTH:  Full-width integer multiplication using Booth's algorithm

        W            = 64                  // Declare a constant
        .text                              // Declare code section
        .align       32                    // Specify desired alignment
        .global      booth                 // Mark entry point for 'booth'
        .proc        booth
booth:
        .prologue                          // Begin procedure 'main'
        .regstk      2,0,0,0               // Declare 2 ins
        .save        ar.lc,r31
        mov          r31=ar.lc             // Save caller's ar.lc
        .body                              // Begin procedure 'booth'
first:  add          r2=W-1,r0;;           // Number of traversals
        mov          ar.lc=r2              // Initialize ar.lc
        mov          r19=0                 // Set bit n-1 to zero
        mov          ret0=in1              // Set R to multiplier
        mov          re1=0;;               // Store first square

cycle:  and          r22=0x1,ret0          // Isolate lowest bit of R
        xor          r23=r19,r22;;         // r23 <- whether to act
        cmp.ne       p6,p0=0,r23           // p6 <- whether to act
        mov          r19=r22               // Bit n - 1 for next iteration

  (p6)  cmp.eq.unc   p7,p8=0,r22           // Add, subtract, or no-op?
  (p7)  add          ret1=ret1,in0         // Add X to L
  (p8)  sub          ret1=ret1,in0;;       // Subtract X from L
        shrp         ret0=ret1,ret0,1      // New R of shifted LR
        shr          ret1=ret1,1           // New L of shifted LR

done:   mov          ar.lc=31              // Restore caller's ar.lc
        br.ret.sptk.many b0;;              // Return to caller
        .endp        booth                 // End procedure `booth`
```

Figure 5: A slightly modified version of the BOOTH function from Evans and Trimper

```
// DECNUM3:  Convert a positive hexadecimal integer into a
             string of ASCII encoded decimal digits

        LEN       = 20                      // Declare a constant
        DOT8      = 0xcccccccccccccccd      // approximately 0.8
        .global   booth                     // External reference to 'booth'
        .data                               // Declare data section
        .align    8                         // Specify desired alignment
X3:     data8     0x123456789               // Number to convert
A3:     .skip     80                        // Storage for ASCII string
STACK:  .skip     LEN                       // User-defined stack

        .text                               // Declare code section
        .align    32                        // Specify desired alignment
        .global   main                      // Mark the mandatory program entry
        .proc     main
main:
        .prologue 12,r32                    // Begin procedure 'main'
        alloc     loc1=ar.pfs,0,6,2,0       // Allocate a new stack frame
        .save     rp,loc0
        mov       loc0=b0                   // Save return address
        .body                               // Begin procedure 'booth'
first:  add       loc2=@gprel(STACK),gp     // loc2 points to STACK
        movl      loc3=DOT8                 // loc3 points to DOT8
        mov       loc5,gp                   // Save global pointer
new:    add       r15=@gprel(X3),gp;;       // r15 points to input number
        ld8       r9=[r15]                  // Load input number
        st1       [loc2]=r0,1;;             // Push zero as a flag

again:  mov       loc4=r9                   // Save previous quotient
        mov       out0=r9                   // arg0 of 'booth' is multiplicand
        mov       out1=loc3                 // arg1 of 'booth' is multiplier
        br.call.sptk.many b0=booth          // Call 'booth', r8:r9 <- out0*out1
        mov       gp=loc5                   // Restore global pointer

nosign: add       r9=r9,loc4;;              // Add X to L
        shr.u     r9=r9,3;;                 // r9 <- quotient = loc4/10
        shladd    r3=r9,2,r9;;              // r3 <- 5*quotient
        add       r3=r3,r3;;                // r3 <- 10*quotient
        sub       r3=r3,r3;;                // r3 <- remainder
        or        r3=0x30,r3;;              // Convert to ASCII
        st1       [loc2]=r3,1               // Store the character
        cmp.ne    p6,p0=r9,r0               // Is quotient non-zero?
   (p6) br.cond.sptk.few again              // Yes, repeat the cycle
        st1       [r16]=r0                  // No, NULL-terminate A3 (stringz)
        add       loc2=1,loc2               // Adjust stack pointer

done:   mov       r8=0                      // Signal completion
        mov       b0=loc0                   // Restore return address
        mov       ar.pfs=loc1               // Restore caller's ar.pfs
        br.ret.sptk.many b0;;               // Return to command line
        .endp     main                      // End procedure `main`
```

Figure 6: A slightly modified version of the DECNUM3 program from Evans and Trimper

12 Program Performance

Performance is typically the driving factor behind the development of new hardware and software designs. Science and engineering applications, where large-scale simulations and time consuming computations dominate, often demand optimal performance from both the low-level hardware and the user-level code.

We now examine the mechanisms provided by the Itanium architecture for software optimization, as well as more general optimization guidelines that can be applied to a variety of applications. We leave hardware optimization considerations to expert sources such as Hennessy and Patterson.

Where possible, we discuss these topics in an implementation-independent manner; however, several of the more advanced topics require at least minor consideration of the architecture's implementation. When this is true, we use the Itanium 2 processor as our guide. For these sections, it may be useful to refer to the details of that implementation, which can be found in Section 13.

12.1 Processor-Level Parallelism

At various points throughout this survey, we have alluded to a feature of modern processor designs called *instruction pipelining*. Pipelining can be likened to an assembly line in a manufacturing process: At each stage of the pipeline, a highly specialized component receives an input from the preceding state, performs one highly specialized function on that input, and produces an output for the next stage. To achieve maximum throughput, each component in the pipeline must perform some sort of useful work at each time-step; this is often referred to as "keeping the pipe full".

Instruction pipelining, then, involves specialized hardware components executing one stage of the instruction cycle. Pipelines have an associated *depth*, which describes the number of stages that perform distinct operations. For example, the instruction pipeline of the Itanium 2 processor involves eight distinct stages; the pipeline's depth is thus eight.

Ideally, each stage performs its operation in the same amount of time as all of the other stages in the pipeline. However, if any one stage takes longer to execute than any other, the steady flow of input-operate-output is temporarily interrupted because at least one stage is left waiting for input; such a situation is known as a *pipeline stall* or *bubble*. We describe several factors that contribute to these stalls shortly.

Three additional terms are often used to characterize instructions in the context instruction pipelining: An instruction is *issued* when it permitted to pass from one stage to the next. The *latency* of a pipeline stage describes the number of time-steps actually required by that stage to perform its operation on an instruction. Finally, an instruction is *retired* when it has passed through the final stage of the pipeline.

We have already mentioned that pipeline stalls can negatively impact the performance of modern CPUs. These stalls typically result from resource conflicts, procedural dependencies, or data dependencies.

Resource conflicts result when instructions in different stages of the pipeline require access to the same area in memory or the same functional unit, for example. Procedural dependencies generally result from branching instructions because partially completed instructions must be halted without altering the state of the machine. Finally, data dependencies typically arise when an instruction requires data that has not yet been computed, loaded, or otherwise made accessible.

The steady flow of input-operate-output in pipelined processors enables maximum performance but can be interrupted by any of the following:

- multiple-issue conflicts,
- branch-induced pipeline flushing,
- producer-consumer dependencies, and
- data stalls from the cache and memory hierarchy.

In EPIC designs, the programmer or compiler must ensure that these situations are avoided whenever possible.

Code	Slot	Unit	Code	Slot	Unit	Code	Slot	Unit	Code	Slot	Unit
0x0	0	M	0x8	0	M	0x10	0	M	0x18	0	M
	1	I		1	M		1	I		1	M
	2	I		2	I		2	B		2	B
0x1	0	M	0x9	0	M	0x11	0	M	0x19	0	M
	1	I		1	M		1	I		1	M
	2	I;;		2	I;;		2	B;;		2	B;;
0x2	0	M	0xa	0	M;;	0x12	0	M	0x1a	0	*
	1	I;;		1	M		1	B		1	
	2	I		2	I		2	B		2	
0x3	0	M	0xb	0	M;;	0x13	0	M	0xb	0	*
	1	I;;		1	M		1	B		1	
	2	I;;		2	I;;		1	B;;		2	
0x4	0	M	0xc	0	M	0x14	0	*	0x1c	0	M
	1	L		1	F		1			1	F
	2	X		2	I		2			2	B
0x5	0	M	0xd	0	M	0x15	0	*	0x1d	0	M
	1	L		1	F		1			1	F
	2	X;;		2	I;;		2			2	B;;
0x6	0	*	0xe	0	M	0x16	0	B	0x1e	0	*
	1			1	M		1	B		1	
	2			2	F		2	B		2	
0x7	0	*	0xf	0	M	0x17	0	B	0x1f	0	*
	1			1	M		1	B		1	
	2			2	F;;		2	B;;		2	

Table 13: Itanium instruction templates

12.2 Instruction-Level Parallelism

Pipelining makes efficient use of the available hardware resources. This technique does not, however, reduce the execution time of any single instruction. Instruction-level parallelism, in which the hardware executes several instructions in parallel, can be used to reduce execution time and achieve greater throughput.

A processor supports *superscalar* execution when it has two or more instruction pipelines that can operate on independent data items in parallel. A superscalar processor fetches, decodes, and (possibly) executes two or more instructions at the same time. Of course, these processors are also susceptible to pipeline stalls, but the additional instruction pipelines typically lead to a performance improvement over a processor with only a single pipeline.

Note that the multiple pipelines in a superscalar processor need not be identical; in fact, multiple specialized pipelines, each handling some subset of an architecture's instruction set, will generally lead to better performance. For example, different integer and floating-point pipelines are common in many modern architectures.

12.3 Explicit Parallelism

Some architectures prescribe that the hardware will direct the execution of program instructions to maximize the available execution resources. Instruction-level parallelism is thus transparent to the programmer. In contrast, EPIC architectures rely on the programmer or compiler to schedule instructions in an intelligent and productive way; this is the "explicitly parallel" part of EPIC designs.

12.3.1 Instruction Templates

In Section 2, we introduced the notion of an instruction bundle: three 41-bit Itanium instructions packaged with a 5-bit instruction template. We noted that the templates provide extra information to the CPU regarding how the instructions contained within the bundle should be executed.

In particular, the template dictates which execution units are required by the instructions in the bundle. The 5-bit field indicates that 32 instruction templates are possible; however, only 24 of these have been defined. Table 13 (previous page) lists the available templates. In this table, "*" indicates that the template code has been reserved for future extensions to the architecture.

Itanium assemblers recognize special directives for manually assigning templates to instruction bundles:

```
{ .mmi                               // use an MMI template for this bundle
  Type M or Type A instruction
  possible stop                      // if M;;MI or M;;MI;;
  Type M or Type A instruction
  Type I instruction
  possible stop                      // if M;;MI;;
}
```

Here, we have illustrated the `.mmi` directive; similar directives are available for the other instruction templates.

You will recall that Type A instructions, the most common type, can be executed by either I- or M-units; this gives programmers and compilers a high degree of latitude when grouping program instructions into bundles.

Typically, instruction templates are not assigned by hand; the compiler or assembler assumes this responsibility. However, for a more thorough discussion, including exercises comparing results from the most popular Itanium compilers, we recommend Evans and Trimper, Section 10.3.1 (pages 302–307).

12.3.2 Data Dependencies and Speculation

Four general cases of data dependencies within an Itanium instruction bundle can be identified:

- Read-after-write (RAW) dependencies are not permitted for Itanium registers (there are a few exceptions) but are permitted for memory: A load from a location in memory to which data has recently been written will retrieve the stored valued.
- Write-after-write (WAW) dependencies are also not generally permitted for registers, but several compare instructions are permitted within the same instruction group. Besides these exceptions, Itanium registers are permitted to occur as a destination operand only once per instruction bundle. Although WAW dependencies are not explicitly forbidden by the architecture, these should be avoided as multiple writes may cause resource conflicts.
- Write-after-read (WAR) dependencies are allowed for both registers and memory.
- Read-after-read (RAR) dependencies are allowed for both registers and memory.

Templates that include the explicit stop must be used for RAW and WAW dependencies involving data from a source register.

The Itanium architecture includes support for *data speculation*. This technique requires special hardware and attempts to minimize the risk or impact of data stalls. These stalls occur when the data needed by an instruction is not yet available in a register because, as we know, a load from memory may take many, many cycles. To mitigate the impact of data stalls, most compilers handle this situation by rearranging the code so that it is logically equivalent to the programmer's intent but with the load appearing earlier in the instruction sequence.

Itanium load instructions that have been moved must fetch the data speculatively. Itanium processors have a special internal structure called the *advanced load address table* (ALAT), which stores a register name and associated memory address. Itanium store instructions query the ALAT, invalidating all entries whose memory address overlaps with any portion of the data to be stored. Data speculations with an invalidated ALAT entry will then fail. Recovery code must also be inserted to handle the situations when a speculative load has failed.

Consider the following example:

```
No Data Speculation        With Data Speculation

                           ld8.a     r20=[r15];;      // Advanced load
<code block A>             <code block A>
st8    [r14]=r24           st8    [r14]=r24
ld8    r20=[r15];;         ld8.c.clr r20=[r15];;      // Check load
add    r20=1,r20           add      r20=1,r20
<code block B>             <code block B>
st8    [r16]=r20           st8    [r16]=r20
```

If the compiler does not know whether registers `r14` and `r15` will point to overlapping regions of memory, then moving the load instruction is a speculative decision and not guaranteed to produce the correct results. This situation situation requires an advanced load (`ld8.a`) with a check load (`ld8.c.clr`) functioning as the recovery routine.

As noted, the load type completer `.a` indicates an advanced load. This instruction inserts an entry for the address contained in register `r15` into the ALAT (possibly displacing some other entry). Later, the check load completer `.c.clr` invokes an ALAT query, searching for register `r20`. If found, the ALAT is cleared and the recovery load is not necessary. However, if the appropriate entry is not found, the load is executed and the value in `r20` is refreshed. The Itanium ISA also provides the `.c.nc` load type completer if the ALAT entry should not be flushed after an ALAT hit.

If the ALAT still holds the necessary entry and no store conflicts have arisen, then the speculation succeeds, minimizing the load latency by executing `<code block A>` while the memory hierarchy satisfies the requested load.

The effectiveness of this speculation depends in part upon the length of `<code block A>`. If the instructions composing this block execute quickly, then a data stall might still occur. The compiler may thus choose a more aggressive rearrangement of the code, as follows:

```
No Data Speculation        With Aggressive Data Speculation

                             ld8.a     r20=[r15];;      // Advanced load
<code block A>               <code block A>
                             add      r20=1,r20
                             <code block B>
st8    [r14]=r24             st8    [r14]=r24
ld8    r20=[r15];;           chk.a.clr r20,recover      // Check load
add    r20=1,r20
<code block B>             back:
st8    [r16]=r20             st8       [r16]=r20
<subsequent code>            <subsequent code>

                           recover:                     // Some other address
                             ld8       r20=[r15];;      // Reload
                             add       r20=1,r20
                             <code block B>
                             br        back
```

In this example, the recovery code (beginning at `recover`) must reload register `r20` and execute the entire instruction sequence (`add` and `<code block B>`) again, if the speculation fails. Such aggressive speculation will lead to large code segments, but can nevertheless prove effective in certain circumstances.

The Itanium ISA provides the `chk.a` instruction to query the ALAT and conditionally branch to the recovery routine. Either `.clr` or `.nc` must be used to complete the instruction; these completers affect the ALAT as described before. Note that while `chk.a` has the same branch range as other IP-relative branch instructions, this instructions executes in an M- and not a B-unit.

Finally, we note that the Itanium ISA provides several forms of an instruction to manually invalidate entries in the ALAT:

```
invala              // Invalidate all ALAT entries
invala.e    r1      // Invalidate entry for general-purpose register r1
invala.e    f1      // Invalidate entry for floating-point register f1
```

This instruction will do nothing if no matching entry is found when a register is specified.

12.3.3 Control Dependencies and Speculation

Control dependency describes situations in which speculative execution is contingent on the logical flow of the program. In these cases, any exceptions that might occur (for example, a floating-point exception) should not be raised if the speculatively executed instructions would not have otherwise been encountered. The NaT bit (for general-purpose registers) and the special NaTVal (for floating-point registers) can suppress this behavior.

In previous sections, we have noted that operations will propagate either the NaT bit or NaTVal if any of its operands are so marked, to be dealt with when convenient. The Itanium ISA includes the `tnat` instruction to test a register's NaT bit:

```
tnat.trel.ctype    pt,pf=r3
```

where the test relationship completer *`trel`* can be `z` (zero) or `nz` (non-zero) and *`ctype`* can be any of those for the compare instructions.

A similar test can be performed for NaTVal in floating-point registers using the `fclass` instruction.

Consider the following example:

No Data Speculation

```
     <code block A>
(px) br.cond notdo
     ld8   r20=[r15];;

     add   r20=1,r20
     <code block B>
notdo:
     <subsequent code>
```

With Control Speculation

```
     ld8.s   r20=[r15];;       // Speculative load
     <code block A>
(px) br.cond notdo
     chk.s   r20,recover
back:
     add     r20=1,r20
     <code block B>
notdo:
     <subsequent code>

recover:                       // Some other address
     ld8       r20=[r15]       // Reload
     br        back;;
```

Here, the load with a long latency depends upon falling through the predicated branch, and the compiler generates a speculative code sequence to overlap execution of other useful work and the load instruction.

The Itanium ISA includes the `chk.s` assembler pseudo-op to branch conditionally if the NaT bit of a register is set. It has the same branch range as other IP-relative branching instructions and can execute in either an I- or M-unit.

It is also possible to combine advanced and speculative loading with the `ld.sa` instruction, causing the ALAT to track success or failure of the load while deferring any exceptions.

12.4 Program Optimizations

Software optimization is considered by many to be a "black art". A good resource outlining general software optimization guidelines is Richard Gerber's book, *The Software Optimization Cookbook*. We also recommend the *Itanium 2 Processor Reference Manual for Software Development and Optimization*, which is part of the Itanium architecture documentation.

12.4.1 Performance Considerations

Several architectural factors may contribute to program's performance. We outline many of these that are of concern for the Itanium architecture. The `lfetch` instruction for software prefetching is also described.

Addressing modes. You will recall from our discussion in Section 2 that the Itanium architecture supports the immediate, register direct, and register indirect addressing modes. For best performance, programs should retain the most frequently used data in processor registers and minimize interaction with system memory. While the cache structures help to alleviate the impact of time-consuming memory operations, it cannot be totally eliminated, and performance degradation can result.

Code size. Code size is much less of a concern in modern architectures than those of even the not-so-distant past. Loop unrolling, which is a common optimization technique employed by modern compilers, may unnecessarily contribute to code bloat. Performance enhancements cannot be guaranteed by the technique but it will always increase the total code size. Loop unrolling can be beneficial, however, due to the relatively high cost of branch instructions.

A typical compiler for the Itanium architecture will offer unrolling as one of many possible methods for loop optimization, but software pipelining using the architecture's rotating register sets will often yield better results. We discuss loop optimization more thoroughly at the end of this section.

Instruction reordering. We encountered instruction reordering previously, when we introduced the Itanium advanced and speculative load instructions. Here, instructions were rearranged so that better performance might result without impacting the logical intent of the program.

At a lower level, instructions within an instruction bundle are mutually independent and can be reordered to enhance the efficiency with which the instructions are executed. This low-level reordering may result in fewer no-op instructions or it may make better use of the available execution units. Typically, Itanium compilers can perform these optimizations but Itanium assemblers cannot.

Inline functions and recursion. You saw in Section 11 that using procedure calls involves a significant amount of overhead. Function inlining, where the body of a procedure or function is inserted directly into the code of the calling routine, can eliminate these overheads at the expense of greater code size.

Software prefetching. In addition to the advanced and speculative loads that we have already discussed, the Itanium ISA includes instructions for prefetching lines into the cache structures:

```
lfetch.lftype.lfhint             [r3]          // mem[r3]
lfetch.lftype.lfhint             [r3],r2       // mem[r3]
                                               // r3 <- r3 + r2
lfetch.lftype.lfhint             [r3],imm9     // mem[r3]
                                               // r3 <- r3 + sext(imm9)

lfetch.lftype.excl.lfhint        [r3]          // mem[r3]
lfetch.lftype.excl.lfhint        [r3],r2       // mem[r3]
                                               // r3 <- r3 + r2
lfetch.lftype.excl.lfhint        [r3],imm9     // mem[r3]
                                               // r3 <- r3 + sext(imm9)
```

where the line containing the address in register `r3` is brought into the cache, followed (optionally) by a postincrement of that address, either by the value in register `r2` or by the 9-bit sign-extended immediate value `imm9`.

A line will be marked *exclusive* when the prefetch instruction includes the `excl` completer; this form is useful when programs will quickly write to an address within the prefetched line.

There are two valid values for *`lftype`*, the line prefetch type completer: none and `fault`. None, which corresponds to omitting this instruction completer, ignores all faults associated with an ordinary load operation, while the `fault` completer will raise these faults as necessary.

There are four valid values for *`lfhint`*, the line prefetch hint completer: none, `nt1`, `nt2`, and `nta`. None, which corresponds to omitting this instruction completer, indicates that the program associates temporal locality in the L1 cache with the prefetched line. The remaining completers, `nt1`, `nt2`, and `nta`, indicate that the program associates nontemporal locality in the L1, L2, or all levels of the hierarchy with the prefetched line.

Other factors. Clearly the discussion of performance factors that we have provide here is not exhaustive. Several other factors, including instruction size, instruction power, and function recursion, are considered more fully by Evans and Trimper (Section 10.6, pages 325–334). We also recommend the sources mentioned previously (Gerber's book and the Itanium documentation) for a wealth of useful information.

12.4.2 Low-level Optimization Hints

The following optimization hints have been adapted from the *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization* and so do not apply to the Itanium architecture in general. The details of the Itanium 2 processor are discussed in Section 13.

Instruction scheduling. Intel offers a number of guidelines that can be used to minimize the chances of implicit stops and other pipeline stalls. We summarize them here:

- Schedule the most restrictive instructions early in a bundle. Doing so will help minimize conflicts with generic instruction subtypes that might otherwise consume the specific port required by a restricted instruction.
- When placing Type A instructions in an instruction bundle's I-slot, try to schedule actual Type I instructions first. Doing so will enable the Type A instructions, which do not require the I-units, to be completed by an M-unit if necessary. Note that it is preferable to place Type A instructions in an M-slot whenever possible.
- Control-speculation (advanced and check) and pair (`ldfps`) floating-point loads require the first two M-units, while other floating-point loads can be performed by any of the Itanium 2 processor's four M-units. When mixing the restrictive and regular floating-point load instructions, schedule the regular loads late in an instruction bundle to ensure that they do not unnecessarily consume the first two M-units and delay the restricted load types.
- Avoid `nop.f`, the floating-point no-op, as unintended floating-point stalls may result from long-latency floating-point instructions.
- Several more dual-issue templates have been added to the Itanium 2 processor. (We discuss this concept in Section 13.) As a result, the `.mfi` instruction template should be avoided.

Branch prediction. We know that branch instructions incur a significant performance impact because they interrupt the sequential flow of program execution. The Itanium architecture offers branch prediction as a means to mitigate the negative impacts.

The branch whether instruction completers were discussed in Section 6. The performance impact of these completers depends on other branch instructions contained within a two-bundle window, as well as other branch information that the processor maintains.

Dynamic branch prediction (`.dpxx`) is the recommended default value. However, if the `ctop` or `cloop` branch type completers are used, static branch prediction (`.spxx`) is recommended.

Static prediction is also recommended for very short (1 or 2 cycle) loops, because loops will not have to wait for the processor to regenerate a new dynamic prediction. If dynamic prediction is used, the processor may stall the loop while it updates the prediction.

Branch prediction hints are not recommended for `.bbb` instruction bundles, as unexpected behavior may result. However, the `.clr` completer can be specified for the slot 0 branch when the use a `.bbb` bundle cannot (or should not) be avoided.

Correctly predicted indirect branches always incur a two-cycle bubble, while the penalty associated with an incorrect result is at least six cycles. To minimize the impact of mispredictions with indirect branch targets, Intel recommends the following:

- Separate the branch register write and indirect branch by at least six access to the L1 instruction cache.
- Add an additional write to the branch register above the actual write as a hint to the target.
- Use different branch registers for each indirect branch instance to minimize conflicts with other indirect branches.

Instruction prefetching. Instruction prefetching, in which instruction cache lines are moved into the L1 instruction cache, is supported by the Itanium 2 processor and is available in two forms: *streaming* and *hint*.

Streaming prefetching corresponds to the use of the `.many` completer for branch instructions and causes the prefetch engine to continuously issue prefetch requests at a rate of one request per cycle. The lines are prefetched from 64 or 128 bytes (depending on the alignment of the branch target) beyond the target address. Streaming prefetching terminates when one of the following conditions is met:

- A predicated-taken branch is encountered.
- A branch misprediction occurs.
- The `brp` instruction is encountered.

The `brp` instruction is used to inform the hardware of an upcoming branch instruction. When used without the `.imp` completer, it is assumed that prefetch engine will have already prefetched beyond the upcoming branch and further prefetches would be useless.

Hint prefetching begins with the `brp` and `mov br` instructions. Several completers for `brp` are available; we recommend Section 8.2 of the *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization* (herein referred to as "the optimization manual") and Section 3:28 of the Itanium Instruction Set Reference for further details of hint prefetching.

We have omitted any discussions of prefetch flushing hints or the `brl` instruction; Section 8.2 of the optimization manual covers the details of these topics.

These low-level hints are of more concern to assembly language programmers or compiler writers than those using a high-level language. However, as we have maintained throughout this survey, understanding the low-level mechanisms employed by an architecture will enable the high-level programmer to write better code. For more details of the topics we have just considered, consult the optimization manual.

12.4.3 Performance Monitoring

The Itanium architecture provides a number of features for advanced performance monitoring, among them the `pmd` registers that were introduced in Section 3. You will recall that the architecture requires that at least eight `pmd` register be implemented. The Itanium 2 processor provides four 48-bit performance counters. In addition, there are over 100 monitorable events and a rich set of advanced monitoring features.

We would certainly stray beyond the intended scope of this work if we were to described all of the performance monitoring features and abilities of the Itanium 2 processor. Instead, we provide an extremely concise overview and refer you to specific pages within Sections 10 and 11 of the the optimization manual for more thorough discussions.

The Itanium 2 processor provides two programming models for performance monitoring: workload characterization and profiling. Both of these models are discussed in Section 10.2 (pages 10-1 through 10-12) of the optimization manual.

The Itanium 2 processor performance monitoring events are broken into several categories, including basic events (clock cycles, for example), branch events, and system events, among others. Using the associated event counters, common performance metrics can be derived, giving the programmer insight into the behavior of the application. These and related topics are taken up in Section 11 of the optimization manual.

12.5 Loop Optimization as a Practical Example

Having at least introduced many of the factors contributing to program performance, we now turn to a practical example: loop optimization. Loops are a common structure provided by high-level languages, and they make many repetitive tasks easy to accomplish with concise and readable code. Because of these features, loop optimization is a important and useful topic to consider.

12.5.1 Loop Unrolling

We mentioned previously that loop unrolling was a optimization technique employed by many compilers. While we noted that this technique may lead to unnecessary code bloat, there are also advantages that can lead to better performance.

Consider an example where the number of registers required by the loop body is significantly less the total number of available processor registers. In this instance, an optimizing compiler will "unroll" the loop; that is, it will duplicate the instructions composing the loop's body some number of times. How many times depends upon a number of factors, including the number of free registers. By doing so, it can reduce the total number of loop traversals and eliminate some amount of overhead. In addition, it may be able to rearrange the instruction sequence more significantly and gain further speed-ups.

There are trade-offs besides code size that must be considered. For example, if the number of loop traversals is not a convenient factor of the number of free processor registers, then special code to handle the remaining traversals must be inserted. Likewise, if that number is determined dynamically and not known at compile-time, then unrolling the loop is significantly more difficult, if not impossible. Finally, as the size of the loop's body increases, there is a greater likelihood that all of its instructions will not fit into the instruction cache. This situation can lead to thrashing, which will negatively impact on the loop's performance.

12.5.2 Software-Pipelined Loops

Often, a better way to handle loop optimization is with software pipelines. You will recall that many modern processors support instruction pipelining at the hardware level. A similar principle can be implemented in software, using the Itanium architecture's support for qualifying predicates, special branch instructions, and *rotating register sets*. We diverge for a moment to introduce the Itanium rotating registers.

Rotating registers. Floating-point registers Fr_{32}–Fr_{127} and predicate registers Pr_{16}–Pr_{63} are designated rotating register sets. General-purpose registers can also operate as rotating registers; they must be `alloc`'d in groups of eight, beginning with Gr_{32}.

Each set of rotating registers will be renamed by incrementing the register number when a special branch instruction is encountered. For example, Fr_{32} becomes Fr_{33}, Fr_{33} becomes Fr_{34}, and so on. Rotating registers allow data contained in a register from a previous iteration to remain accessible in subsequent iterations, but in a differently named register; each new iteration is given a new group of registers with which to operate.

Note that the hardware automatically handles rotation and renaming of all three rotating register sets.

Modulo-scheduling. Now, using a rotating register set, programmers are able to create a software pipeline and *modulo-schedule* the instructions in a loop, enabling instructions from different iterations of the loop to execute in parallel.

Modulo-scheduled loops typically consist of three phases: the prolog phase, the kernel phase, and the epilog phase. During the prolog phase, the software pipeline fills. Some instructions of the loop will be predicated false.

During the kernel phase, a new iteration of the loop will begin with each cycle, while some previous iteration completes. Here, the software pipeline is full, and generally all of the instructions are predicated true.

Finally, in the epilog phase, the software pipeline drains; no new iterations are started and the uncompleted iterations are finished. With each cycle, one predicate register will be set to false.

Branch instructions for software pipelining. The Itanium ISA provides special branch instructions for loop control that are used with rotating registers:

```
        br.ctop.bwh.ph.dh     target      // rotate and return to top
                                          // exit when ar.lc = 0 and
                                          // ar.ec = 1
        br.cexit.bwh.ph.dh    target      // rotate and fall through
                                          // exit when ar.lc = 0 and
                                          // ar.ec = 1

  (qp)  br.wtop.bwh.ph.dh     target      // rotate and return to top
                                          // exit when qp = 0 and
                                          // ar.ec = 1
  (qp)  br.wexit.bwh.ph.dh    target      // rotate and fall through
                                          // exit when qp = 0 and
                                          // ar.ec = 1
```

where the valid values for *bwh*, *ph*, and *dh* are the same as those for the regular branch instructions.

The first two forms (`br.ctop` and `br.cexit`) are used for counted loops. We have already encountered the Itanium loop count register `ar.lc`. The values in this register, and in `ar.ec`, the epilog count register, are used to control register rotation and the program's flow of execution.

During the prolog and kernel phases, the value of `ar.lc` will be greater than zero. In this case, the `br.ctop` and `br.cexit` instructions will decrement `ar.lc`, set predicate register `p63` to one, and rotate all three register sets. So, for the next iteration, `p16` will contain a one because `p63` was rotated "into" `p16`.

Then, during the epilog phase, the value of `ar.lc` will be zero, and `ar.ec` will be greater than one. Here, the `br.ctop` and `br.cexit` instructions will decrement `ar.ec`, set predicate register `p63` to zero, and rotate all three register sets. So, for the next iteration, `p16` will contain a zero because `p63` was rotated "into" `p16`.

Finally, at the end of the epilog, `br.ctop` will fall through instead of branching and `br.cexit` will branch to `target` rather than fall through.

The following schematic demonstrates register rotations for counted loops:

Pipeline Phase	ar.lc	ar.ec	p63
Prolog	decremented	unchanged	1
Kernel	decremented	unchanged	1
Epilog	0	decremented	0

Upon exit, `ar.lc`, `ar.ec`, and `p63` will typically be zero.

The last two forms (`br.wtop` and `br.wexit`) are used for while loops. These instructions operate similarly to those for counted loops; however, the qualifying predicate register `qp` and the values in `ar.ec` control register rotation and the program's flow of execution.

During the prolog and kernel phases, the value of `qp` will be one. In this case, the `br.wtop` and `br.wexit` instructions will set predicate register `p63` to zero and rotate all three register sets. So, for the next iteration, `p16` will contain a zero because `p63` was rotated "into" `p16`.

After the value of `qp` becomes zero, but while `ar.ec` is greater than one (the epilog phase), the `br.wtop` and `br.wexit` instructions will decrement `ar.ec`, set predicate register `p63` to zero, and rotate all three register sets. So, for the next iteration, `p16` will contain a zero because `p63` was rotated "into" `p16`.

Finally, at the end of the epilog, `br.ctop` will fall through instead of branching and `br.cexit` will branch to `target` rather than fall through.

The following schematic demonstrates register rotations for counted loops:

Pipeline Phase	qp	ar.ec	p63
Prolog	1	unchanged	0
Kernel	1	unchanged	0
Epilog	0	decremented	0

Upon exit, `qp`, `ar.ec` and `p63` will typically be zero.

12.5.3 Writing a Software Pipelined Loop

To illustrate software pipelined loops, we again use (slightly modified) example code from Evans and Trimper. Like the `DOTCLOOP` program in Section 10, the code in Figure 7 (page 64) computes the scalar product of two vectors. However, unlike the previous version, this example uses the special branch instructions, rotating register sets, and qualifying predicates to modulo-schedule the program's main loop.

This code is written for the Itanium 2 processor. While it will run on any Itanium implementation, it achieves optimal performance on this particular implementation because it accounts for instruction latencies and other factors that are specific to the Itanium 2 processor.

Evans and Trimper also include an implementation-independent version of this program (`DOTCTOP`) in Section 10.5.1 (pages 316–321); see their book for details.

Determining the rotating register sets. First, we begin by recognizing that an integer load (from the L1 cache) has a minimum latency of two cycles when the data are to be operated on by the `pmpy2.r` instruction, which itself has a latency of 3 cycles. Now, we can outline the software pipeline for the program's main loop, using a three-element vector as an example, as follows:

Cycle	Stage 0	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5	Stage 6
0	ld2(1)						
1	ld2(2)						
2	ld2(3)		pmpy(1)				
3			pmpy(2)				
4			pmpy(3)				
5						sxt(1)	
6						sxt(2)	add(1)
7						sxt(3)	add(2)
8							add(3)

where stage 1 is a no-op to account for the extra load latencies, and stages 3 and 4 are also no-ops, accounting for the latency of the `pmpy2.r` instruction used in the program.

Using this pipeline, the elements of vector X are valid in stages 0–2; thus, three rotating registers are required and we use `r32`, `r33`, and `r34`. The elements of vectors X and Y are valid in stages 2–5. By reallocating `r34` as the destination in stage 2, only three more rotating registers are required; we use `r35`, `r36`, and `r37`. Results from the sign-extend instruction are valid in stages 5 and 6, but `r37` can be reallocated as the destination in stage 5, so only one new register is required (`r38`). Finally, the elements of vector Y will occupy three additional registers; we use `r39`, `r40`, and `r41`. A total of ten rotating registers, `r32–r41`, are required.

Now we consider the rotating predicate registers that will be used to control the flow of execution within the software pipeline. Note that only two stages are "active" during any given execution cycle. Four predicate

registers (`p16`, `p18`, `p21`, and `p22`) will be assigned to the operational pipeline stages, while `p17`, `p19`, and `p20` correspond to the no-op stages.

The loop count register (`ar.lc`) should be zero at the cycle after which `p63` will become zero, making `p16` zero on the following cycle. This requirement typically implies that `ar.lc` should be set to one less than the number of traversals required by the loop. The epilog counter (`ar.ec`) should be set to seven because there are seven pipeline stages to be drained. The following schematic demonstrates this behavior:

Cycle	ar.lc	ar.ec	p16	p17	p18	p19	p20	p21	p22
0	2	7	1	0	0	0	0	0	0
1	1	7	1	1	0	0	0	0	0
2	0	7	1	1	1	0	0	0	0
3	0	6	0	1	1	1	0	0	0
4	0	5	0	0	1	1	1	0	0
5	0	4	0	0	0	1	1	1	0
6	0	3	0	0	0	0	1	1	1
7	0	2	0	0	0	0	0	1	1
8	0	1	0	0	0	0	0	0	1

Note that the loop counter should stop on the value zero, while the epilog counter should stop on the value one. In addition, the number of software pipeline cycles is `ar.lc` + `ar.ec`; here, 2 + 7 = 9.

Optimizing the instruction schedule. Next, consider the order of instructions that compose the main loop in the original `DOTCLOOP`:

```
top:
          ld2         r21=[r14],2
          ld2         r22=[r15],2;;
          pmpy2.r     r21=r21,r22
          sxt4        r21=r21;;
          add         r20=r20,r21
          br.cloop.sptk.few top;;
```

Using this same order in the software pipelined version of the program, three instruction bundles are necessary:

```
top:
   (p16)  ld2         r32=[r14],2   // M
   (p16)  ld2         r39=[r15],2   // M
   (p17)  pmpy2.r     r34=r34,r41   // I

          nop.m                     // M
   (p18)  sxt4        r37=r37       // I
   (p19)  add         r20=r20,r38   // I


          nop.m                     // M
          nop.f                     // F
          br.ctop.sptk.few top;;    // B;;
```

You will note that some explicit stops have been eliminated, which is possible because the instructions in the software pipelined loop are predicated. Nevertheless, even if we ignore the delays of loading data from

memory and the actual latencies of instructions on a specific implementation, then this schedule requires at least two cycles per iteration.

By taking advantage the characteristics of Type A instructions, we can switch the order of the `sxt4` and `add` instructions to condense the sequence into two instruction bundles:

```
top:
  (p16)   ld2         r32=[r14],2   // M
  (p16)   ld2         r39=[r15],2   // M
  (p17)   pmpy2.r     r34=r34,r41   // I

  (p19)   add         r20=r20,r38   // M

  (p18)   sxt4        r37=r37       // I
          br.ctop.sptk.few top;;    // B;;
```

resulting in a smaller code size and better instruction cache usage. In addition, execution time on an Itanium 2 processor will be reduced because it has four M-units. Note that the explicit stop after the `br.ctop` will ensure the proper logical ordering from cycle to cycle.

An analysis of this optimized instruction schedule by Evans and Trimper shows that the minimum number of cycles is 13, while the original version requires 21 cycles, indicating a significant savings for the optimized, modulo-scheduled loop. (The details of this analysis can be found in Section 10.5.1, page 319 of Evans and Trimper.)

Modulo-scheduling and software pipelining are powerful optimization techniques, but they can be confusing. We recommend a careful review of these topics, particularly the `DOTCTOP2` code in Figure 7.

```
// DOTCTOP2:  Compute the scalar product of two vectors

        N         = 3                    // Declare a constant
        .data                            // Declare data section
        .align    8                      // Specify desired alignment
P:      .skip     8                      // To store the product
X:      data2     -1,+3,+5               // First vector of 16-bit values
Y:      data2     -2,-4,+6               // Second vector of 16-bit values

        .text                            // Declare code section
        .align    32                     // Specify desired alignment
        .global   main                   // Mark mandatory 'main' program entry
        .proc     main
main:
        .prologue                        // Begin prologue section
        .save     ar.lc,r9
        mov       r9=ar.lc;;             // Save caller's ar.lc
        .body                            // Begin procedure 'main'
first:  alloc     r10=ar.pfs,0,16,0,16   // Allocate a new stack frame
        movl      r14=X                  // Gr14 = pointer to X
        movl      r15=Y                  // Gr15 = pointer to Y
        movl      r16=P                  // Gr16 = pointer to P
        mov       r20=0                  // r20 = scalar product
        mov       ar.lc=N-1              // Initialize ar.lc
        mov       ar.ec=7                // Initialize ar.ec
        mov       pr.rot=0x10000;;       // Initialize predicates

top:
  (p16) ld2       r32=[r14],2            // Load element, increment pointer
  (p17) ld2       r39=[r15],2            // Load element, increment pointer
  (p18) pmpy2.r   r34=r34,r41            // Multiply elements
  (p22) add       r20=r21,r38            // Update scalar product
  (p21) sxt4      r37=r37                // Sign-extend result to 64 bits
        br.cloop.sptk.few top            // More elements to process?
        st8       [r16]=r20              // No, store the scalar product

done:   mov       ret0=0                 // Signal completion
        mov       ar.lc=r9               // Restore caller's ar.lc
        mov       ar.pfs=r10             // Restore caller's ar.pfs
        br.ret.sptk.many b0;;            // Return to command line
        .endp     main                   // End procedure 'main'
```

Figure 7: A slightly modified version of the DOTCTOP2 program from Evans and Trimper

13 Itanium Implementations

As of this writing, Intel has marketed two processor implementations of the Itanium architecture: the (original) Itanium processor and the Itanium 2 processor. In addition, the Hewlett-Packard Company (HP) has developed a software-based Itanium ISA simulator, called Ski. In the final section of this survey, we discuss the more relevant details of each implementation.

13.1 The Itanium-Family Processors

The first "market-ready" implementation of the Itanium architecture was, not surprisingly, Intel's Itanium processor. The life-cycle of the original Itanium processor was relatively short, for a number of reasons that we do not discuss. Only one year later, Intel released the Itanium 2 processor, which includes a number of enhancements over the original implementation.

Table 14 (next page) compares many characteristics of the two Itanium processors. We elucidate only the important details, largely for comparative purposes.

13.1.1 Cache Hierarchy

The Itanium processors include three distinct levels in the cache hierarchy, differing in terms of size, speed, and physical location. The cache structure "closest" to the processor is the level 1 (L1) cache. The Itanium processors divides this cache into two regions, one for instructions (L1-I) and one for data (L1-D), following the *Harvard memory architecture*. The L1-I cache is read-only, while the L1-D cache is read-write. Each cache structure has a separate connection to the CPU.

In contrast, the level 2 (L2) and level 3 (L3) cache structures use a *von Neumann memory architecture*, where no distinction is made between instructions and data. The L3 cache, which is not a common feature of most contemporary architectures, functions as a *backside cache*, because it connects to the processor using a separate bus. The L3 cache monitors L2 activity and mimics the data access patterns while retaining a longer history of that activity due to its larger size.

Figure 8 shows the structure of the cache hierarchy for the Itanium processors.

In general, the cache structures of the Itanium processor are smaller and slower than those of the later Itanium 2 processor. Although the L3 cache of an Itanium processor can be larger than that of an Itanium 2, access times are slower because the cache is merely in-package, rather than on-chip as it is in the Itanium 2 processor. The original Itanium processor exhibits fixed load latencies (ranging from 2–24 cycles, depending on the cache level and data type), whereas this quantity varies from 1–18 cycles in the Itanium 2 because of possible access conflicts.

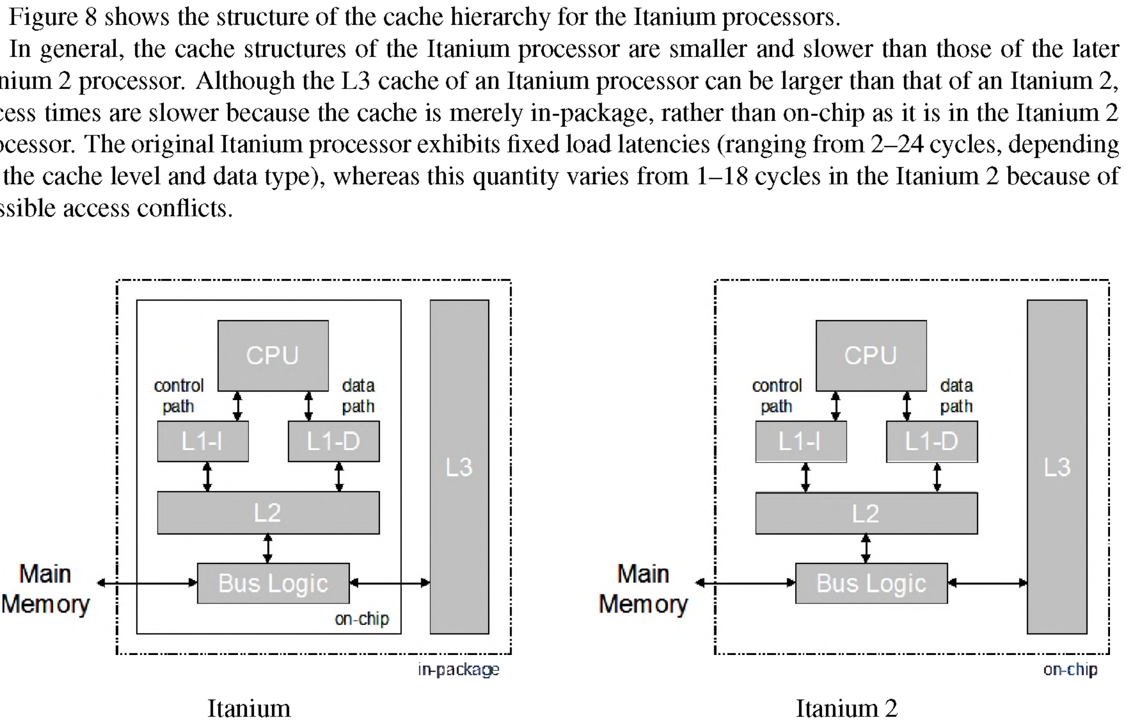


Figure 8: Structure of the cache hierarchy for the Itanium processors

Characteristic	Itanium	Itanium 2
Development code name	Merced	McKinley
Year of market release	2001	2002
Chip technology		
CPU speeds	733 MHz, 800 MHz	900 MHz, 1.0 GHz, 1.3 GHz, 1.4 GHz, 1.5 GHz
Process feature size	180 nm	180 nm
Number of transistors	$25 * 10^6$	$222 * 10^6$
Number of layers	6	6
Operating voltage	1.5 V	1.5 V
Power consumption	116–130 W	130 W
Processor features		
Physical stacked registers	96	96
RSE modes	Enforced lazy	Enforced lazy
Integer units	2 M-units, 2 I-units	4 M-units, 2 I-units
Memory units	2 load, 2 store	2 load, 2 store
Parallel floating-point units	2	1
Issue ports	9	11
Pipeline depth		
Integer	10	8
Floating-point	12	10
Memory support		
Physical address bits	44	50
Virtual address bits	54	64
Data bus width	64 bits	128 bits
Maximum page size	256 MB	4 GB
System bus		
Speed	266 MHz	400 MHz
Width	64 bits	128 bits
Bandwidth	2.1 GB/s	6.4 GB/s
L3 Cache		
Size	2 MB, 4 MB	3 MB, 4 MB, 6 MB
Location	off-chip, in-package	on-chip

Table 14: Characteristics of the current Itanium processors

Benchmarks have shown that the Itanium 2, which exhibits several optimizations including those in the cache hierarchy, achieves approximately twice the instruction throughput of the original Itanium processor, even when running with clock speeds that are less than a factor of two faster.

Table 15 (next page) summarizes the relevant differences of the Itanium and Itanium 2 cache structures.

13.1.2 Execution Units and Issue Ports

Table 14 shows that the processors also differ in the number and kinds of execution units. The Itanium 2 adds two more M-units and two additional instruction issue ports.

You will recall that Type A instructions can execute in either M- or I-units; the additional M-units raise the superscalar degree for these instructions to six. Furthermore, many more pairings of instruction bundle templates can be issued in a given clock cycle. Table 16, in which the row represents the first bundle of the

Processor	Cache Level	Total Size	Line Size	Load Latency
Itanium				
	L1-I	16 KB	32 B	2 cycles
	L1-D	16 KB	32 B	2 cycles
	L2	96 KB	64 B	6 cycles (integer) 9 cycles (fbating-point)
	L3	2 MB, 4 MB	64 B	21 cycles (integer) 24 cycles (fbating-point)
	Physical memory	$\leq$ 16 TB		$>$ 100 cycles
Itanium 2				
	L1-I	16 KB	64 B	1 cycle (instruction)
	L1-D	16 KB	64 B	1 cycle (integer)
	L2	256 KB	128 B	5–9 cycles (integer) 6–10 cycles (fbating-point) 7–11 cycles (instruction)
	L3	3 MB, 4 MB, 6 MB	128 B	12–16 cycles (integer) 13–17 cycles (fbating-point) 14–18 cycles (instruction)
	Physical memory	$\leq$ 1 PB		$>$ 100 cycles

Table 15: Characteristics of the Itanium and Itanium 2 cache structures

pair and the column, the second, captures the possible dual-issue pairs for each processor.

	MII	**MLX**	**MMI**	**MFI**	**MMF**	**MIB**	**MBB**	**BBB**	**MMB**	**MFB**
MII	I2	none	I2	I2	I2	I2	both	both	I2	both
MLX	I2	I2	I2	both	I2	both	I2	both	I2	both
MMI	I2	I2	I2	I2	I2	I2	I2	both	I2	I2
MFI	I2	both	I2	both	I2	both	both	both	I2	both
MMF	I2	I2	I2	I2	I2	I2	I2	both	I2	I2
MIB	I2	both	I2	both	I2	both	both	none	I2	both
MBB	none	none	none	none	none	none	none	none	none	none
BBB	none	none	none	none	none	none	none	none	none	none
MMB	I2	I2	I2	I2	I2	I2	I2	I2	none	I2
MFB	both	both	I2	both	I2	both	both	none	I2	both

Table 16: Possible dual-issue instruction bundles for the Itanium and Itanium 2 processors

Note the significantly higher number of cells labeled "I2", indicating those pairings that only the Itanium 2 processor supports. The additional M-units and issue ports significantly impact the degree of possible parallelism in the Itanium 2 processor.

13.1.3 Pipelines

Table 14 also shows that the implementations differ in the length of their execution pipelines. The pipeline stages of each processor are described in Table 17.

Processor	Stage	Mnemonic	Description
Itanium			
	1	IPG	Generate instruction pointer
	2	FET	Prefetch up to 6 instructions per cycle; predict branch direction
	3	ROT	Rotate instructions of current group into position; calculate branch address
	4	EXP	Issue up to 6 instructions through 9 ports
	5	REN	Rename registers
	6	WLD	Deliver data loaded from memory, requires latency of at least one cycle
	7	REG	Deliver data from the Gr, Fr, and Pr registers
	8	EXE	Execute operations
	9	DET	Detect exceptions; abandon results if predicate register was false
	10	WRB	Store results in Gr, Fr, and Pr registers, as necessary
Itanium 2			
	1	IPG	Generate instruction pointer
	2	ROT	Rotate instructions of current group into position
	3	EXP	Issue up to 6 instructions through 11 ports
	4	REN	Rename registers; decode instructions
	5	REG	Deliver data from the Gr, Fr, and Pr registers
	6	EXE	Execute operations
	7	DET	Detect exceptions; abandon results if predicate register was false; correct mispredicted branches
	8	WRB	Store results in Gr, Fr, and Pr registers, as necessary

Table 17: The Itanium and Itanium 2 pipelines

In the original Itanium processor, each stage belongs to one of four larger phases: the front-end (IPG, FET, ROT); instruction delivery (EXP, REN); operand delivery (WLD, REG); and execution, with a change of machine state (EXE, DET, WRB).

However, in the Itanium 2 processor, there are only two larger phases: the front-end (IPG, ROT) and the back-end (EXP, REN, REG, EXE, DET, WRB).

It should be obvious that we have not attempted an exhaustive comparison of the two Itanium family processors. Several more details of these processor are available from the Intel web site (http://www.intel.com/).

13.2 The Ski Simulator

HP maintains a freely available software-based Itanium ISA simulator, called Ski. The simulation environment executes on IA-32 architectures running the Linux operating system. Ski functionally simulates the Itanium instruction set architecture, not a specific processor implementation. As a result, it is extremely fast. However, the Ski simulator cannot be used to measure the actual performance of a simulated program because it does not simulate the micro-architectural characteristics of an Itanium implementation.

The HP Ski web site states that the simulator is well-suited for:

- Itanium application development on non-native hardware,
- Itanium compiler tuning,
- operating system and firmware development for Itanium architectures, and
- functional instruction verification of Itanium processor implementations.

The simulator provides two execution modes: system-mode, in which both the system-level and the application-level instructions are simulated, and user-mode, in which only the application level instructions are simulated. While user-mode simulation is faster, it does not support many useful or necessary features, multi-threading being a prime example. All system calls are intercepted and translated into calls for the host machine's operating system. In system-mode, applications execute along with an actual Itanium Linux kernel. For better simulation accuracy, system-mode execution is required.

HP also provides the Native User Environment (NUE), which emulates the IA-64 Linux environment and is intended for use with the Ski simulator. NUE provides the compiler, linker, assembler, libraries, and execution environment necessary to develop IA-64 Linux applications. We note that the compiler included with NUE is not an optimizing compiler, but such a compiler is available from SGI, Inc. See SGI's web site (http://oss.sgi.com/projects/Pro64/) for more information.

We have found the Ski simulator and NUE to be extremely useful for porting existing applications to the Itanium architecture. Information about obtaining and installing the Ski and NUE software can be found at HP's IA-64 Linux Developer Tools web site (http://www.software.hp.com/products/LIA64/).

Bibliography and Additional Resources

Evans, James S. and Gregory L. Trimper, *Itanium Architecture for Programmers*, Upper Saddle River, NJ: Prentice Hall, Inc., 2003.

Gerber, Richard, *The Software Optimization Cookbook*, Hillsboro, OR: Intel Press, 2002.

Hennessy, John L. and David A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. San Francisco, CA: Morgan Kaufmann Publishers, Inc., 2002.

Hennessy, John L. and David A. Patterson, *Computer Organization and Design: The Hardware/Software Interface*, 2nd ed. San Francisco, CA: Morgan Kaufmann Publishers, Inc., 1997.

Hewlett-Packard Company, "HP IA-64 Linux Simulator (Ski)", IA-64 Linux Developer's Kit web site. Available at http://www.software.hp.com/products/LIA64/overview1a.htm.

Hewlett-Packard Company, "IA-64 Linux Developer's Kit (CD)", IA-64 Linux Developer's Kit web site. Available at http://www.software.hp.com/products/LIA64/overview4a.htm.

Hewlett-Packard Company, "IA-64 Root File System", IA-64 Linux Developer's Kit web site. Available at http://www.software.hp.com/products/LIA64/overview3a.htm.

Hewlett-Packard Company, "Native User Environment (NUE)", IA-64 Linux Developer's Kit web site. Available at http://www.software.hp.com/products/LIA64/overview2a.htm.

Intel Corporation, "Application Architecture", revision 2.1, *Intel Itanium Architecture Software Developer's Manual*, Vol. 1, 2002.

Intel Corporation, "Instruction Set Reference", revision 2.1, *Intel Itanium Architecture Software Developer's Manual*, Vol. 3, 2002.

Intel Corporation, "System Architecture", revision 2.1, *Intel Itanium Architecture Software Developer's Manual*, Vol. 2, 2002.

Intel Corporation, *Itanium 2 Processor Reference Manual for Software Development and Optimization*, 2003.

Intel Corporation, *Itanium Architecture Assembler User's Guide*, 2000.

Markstein, Peter, *IA-64 and Elementary Functions: Speed and Precision*. Upper Saddle River, NJ: Prentice Hall PTR, 2000.

Rau, B. Ramakrishna, "Dynamic Scheduling Techniques for VLIW Processors," *HP Technical Report HPL-93-52* (June 1993). Available at http://www.hpl.hp.com/techreports/.

Rau, B. Ramakrishna, and Joseph A. Fisher, "Instruction-Level Parallel Processing: History, Overview, and Perspective," *HP Technical Report HPL-92-132* (October 1992). Available at http://www.hpl.hp.com/techreports/.

SGI, Incorporated, "Pro64", Developer Central Open Source web site. Available at http://oss.sgi.com/projects/Pro64/.