

Chapter 1

Introduction

CISC, RISC, VLIW, and EPIC Architectures

The first commercial microprocessor was the Intel 4004, introduced by Intel Corporation more than 30 years ago, in 1971. This 4-bit processor was followed by 8-bit, 16-bit, 32-bit, and 64-bit processors from Intel Corporation and other manufacturers. The most successful family of processors started with the 16-bit Intel[®] 8086, which set the basis for the Intel[®] x86 architecture (also referred to as IA-32).

CISC and RISC

The x86 processors belong to the CISC (Complex Instruction Set Computing) category, mainly because their instructions may be quite complex or have variable length. They use a relatively small number of registers, and are capable of accessing memory locations directly. Complex instructions are sequenced in microcode in modern CISC processors.

A different line, derived from CISC, is represented by the RISC (Reduced Instruction Set Computing) processors introduced in the 1980s, which are characterized mostly by how they differ from CISC processors. The instructions are of fixed length, and of a regular format. Operations are performed on registers only, of which a larger number are available than on CISC processors. The only memory operations are load and store. The hardware in RISC processors is simpler in principle

than in CISC ones, because a RISC architecture relies more on the compiler for sequencing complex operations.

Advances in both CISC and RISC processors were possible on one hand through technological progress in the manufacturing area, and on the other through new processor architecture and microarchitecture features.

Technological advances in manufacturing, driven by increasing demand for computing power, generally followed Moore's Law. Named after Intel Corporation co-founder Gordon Moore, this states that the number of transistors in a microprocessor doubles approximately every 18 months. These advances combined with new circuit design techniques also led to steadily rising clock frequencies. Individual operations can thus be executed more quickly and/or with lower power requirements on each new generation of processors.

On the processor microarchitecture side, the most notable improvements were brought by the introduction of multiple execution units, pipelined execution units, and out-of-order execution.

Today, CISC processors dominate the personal computer market and represent a significant fraction of the low- and mid-range servers and workstations. RISC processors are found mostly in workstations and servers.

Parallel Execution

The presence of multiple functional units is the main characteristic of *multiple-issue* processors, where more than one instruction can be issued (started) during a given clock cycle. Since they can operate on more than one set of input operands in parallel, such processors are often said to be *superscalar*. Even without multiple execution units, the technique of *pipelining* effectively adds more parallelism. Pipelined execution units are common for instructions with a long execution time, such as floating-point multiplication. A pipelined operation is split into several stages, each performed by a different hardware subsystem, so that the early stages of one operation can be overlapped with the later stages of a previous one, in a technique analogous to a factory production line. If an operation is fully pipelined in n stages, then n distinct operations can be in progress at any time, leading to much higher *throughput* than a non-pipelined implementation. If an instruction takes one clock cycle for each stage, then a result can be generated every clock cycle, even though the instruction latency is of n clock cycles. Figure 1.1 shows a five-stage execution pipeline as an example. When there is a steady flow of instructions through the pipeline, all

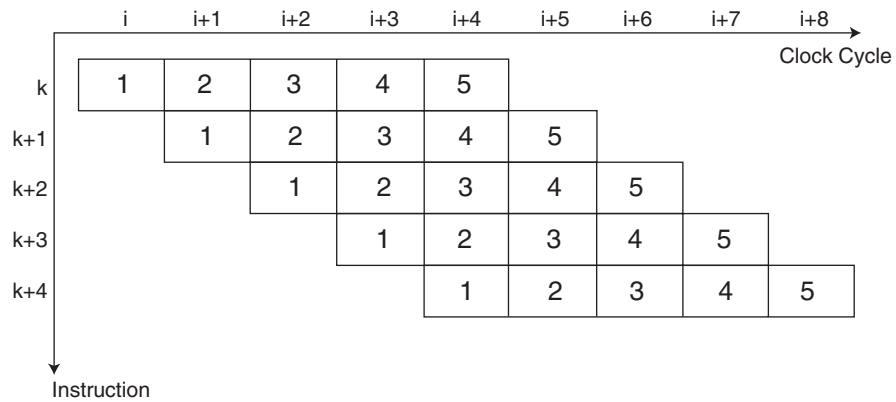


Figure 1.1 Example of Pipelined Execution Unit

stages are being executed concurrently for successive instructions; for example stage 2 of instruction k is being executed at the same time as stage 1 of instruction $k+1$, and so on.

Although most modern processors are capable of executing several instructions in parallel, it is not always easy to exploit this feature in real programs, which are usually written in an explicitly serial fashion. Mainstream high-level languages such as C and FORTRAN allow a limited freedom for the implementation to execute operations in parallel or in arbitrary order. Ultimately, programs need to be compiled into machine code, and most conventional instruction sets do not allow for the indication of parallel execution. In order to exploit the ability to perform multiple instructions in parallel, a processor must reconstruct the implicit parallelism in the code from the serialized instruction stream, which constitutes a complicated process.

VLIW

To overcome the difficulty of finding parallelism in machine-level object code, other processor architectures were developed. Among these, VLIW (Very Long Instruction Word) has a special significance because of its legacy. In a VLIW processor, multiple instructions are packed together and issued in parallel to an equal number of execution units. The task of checking that there are no unwanted dependencies between instructions that are executed in parallel belongs to the compiler rather than the processor, leading to compiler-based superscalar

technology. The hardware in VLIW processors was simpler, but the drawback was that two consecutive generations of the same VLIW processor family were not compatible at the binary level. Code had to be recompiled every time a new processor was introduced.

While not a commercial success, VLIW architectures influenced other modern superscalar and out-of-order superscalar machines. The difference is that in the latter the hardware rather than software attempts to find and exploit parallelism in the instruction sequence.

EPIC

The performance of processors continues to increase. But the physical limits for the manufacturing technology will eventually be reached, rendering Moore's Law inapplicable. Substantial further advances can be attained only by allowing a processor to operate on more bits at a time, and to execute more instructions in parallel. This was the motivation that led to the design of the Itanium[®] processor family. Based on the EPIC (Explicitly Parallel Instruction Computing) design philosophy, the Itanium architecture was co-developed by Intel Corporation and Hewlett-Packard Company, combining the best in the RISC and VLIW architectures, also adding several features that resulted from recent research studies in processor architecture. The result is a processor architecture that can handle a large amount of work based on its ability to feed instructions quickly to several execution units.

The basic EPIC principle is that the programmer or compiler should be able to indicate the inherent parallelism of programs *explicitly* in the instruction sequence, rather than obliging the processor to reconstruct it from a particular sequence of serial operations [1]. Relying more on the compiler or performance programmer to specify parallelism means that much of the out-of-order execution and dependency-check logic found in other processors is not necessary anymore. Instead, the saved silicon space can be used for more functional units and more registers that enable execution of several instructions in parallel.

To date, two implementations of the Itanium architecture have been introduced by Intel Corporation. The Itanium processor provided hardware manufacturers and software writers with a first development vehicle. The second implementation, represented by the Itanium 2 processor, increases the performance level of the Itanium processor by a factor of 1.5 to 2 in several cases. With microcode or software support, both implementations are backwards-compatible with the Intel x86 and the Hewlett-Packard PA-RISC families of processors.

64-Bit Computing

The transition from 32-bit computing to 64-bit computing is gradual, will take a long time, and will probably never be complete, as many applications will still run optimally on 32-bit systems. 64-bit processors started being developed and used over the last decade and a half, when it became clear that high-end applications could use more address space and computing power than those offered by 32-bit processors. The most notable general-purpose 64-bit RISC processors of this period were the Intel 80860, the UltraSPARC[†] family of processors from Sun Microsystems, the MIPS processors from MIPS Technologies, the Alpha processor family from Digital Equipment Corporation, the Power processors from IBM, and the PA-RISC family of processors from Hewlett-Packard Company. These are followed today by the Intel[®] Itanium processor family, developed by Intel Corporation in cooperation with Hewlett-Packard Company to take 64-bit computing performance to new levels. Itanium processors target the most demanding enterprise and high-performance computing applications, addressing the growing needs for data communications, storage, analysis, and security, while also providing performance, scalability, and reliability advantages at significantly lower costs than before. The designers of this architecture believe that its implementations will establish new standards of price and performance for 64-bit computing.

What are the benefits of 64-bit computing? First, increasing the addressing capability from 32 to 64 bits increases the amount of memory that can be addressed directly by a process from 4 gigabytes to 16777216 terabytes (or 16 exabytes), allowing more and larger data structures to be accessed quickly from memory, rather than disk. While millions of terabytes are more than adequate for today's needs, 4 gigabytes are actually insufficient for an increasing number of applications. Relatively common 32-bit desktop systems of today already use 1 gigabyte of memory or more. Second, increasing the size of data items from 32 bits to 64 bits allows one to process twice as much information per clock cycle. Internal and external data buses are wider, increasing the communication bandwidth between the various subunits of the processor, and between processor and the memory hierarchy.

Which applications will benefit from using 64-bit, and in particular Itanium-based computing systems? Common desktop applications have no immediate need for the computing power or addressing capabilities of a 64-bit processor, but an increasing number of mid-range and high-end applications already do, or will soon, require such capabilities.

These are mainly programs that demand a lot of memory space and/or perform a large amount of computation. Examples include applications accessing large databases or delivering Internet content, programs that use 64-bit long integers, and data-intensive applications solving scientific and engineering problems.

Database applications can use the large memory capabilities of 64-bit Itanium-based servers to reduce access times to data items. These applications can also benefit significantly from the increased integer and floating-point computing power of the Itanium architecture for running encryption/decryption algorithms that today take a large fraction of the execution time for secure transactions on other systems. Network management and security for Internet content delivery applications, as well as secure on-line transactions are increasingly demanding. These applications will benefit from using Itanium processors, since data and operations need to be validated in real-time.

Several 32-bit compilers were already supporting 64-bit integers. The transition to 64-bit systems will just make applications using this data type run much faster than before.

Scientific and engineering applications can also take advantage of the increased floating-point performance of Itanium processors. These applications include among others quantum chromodynamics (QCD), quantum mechanics, molecular simulation, cell research, or new drug discovery applications, computer-aided design tools, and solvers for large equation systems used in a variety of scientific and technical problems. Digital content creation applications that require high bandwidth, large memory, and powerful floating-point performance are also going to benefit from running on Itanium processors. Such applications can run very slowly on workstations based on 32-bit processors because of the smaller data item size, and also because of the continuous data traffic between storage disks and the memory system. Reduced swapping between memory and disk on Itanium-based systems are likely to increase performance of some applications by up to two orders of magnitude.

Software simulators and then hardware systems based on Itanium processors were used to develop software for the new architecture, starting with 64-bit compilers and operating systems, and ending with porting major applications to Itanium-based platforms. The most widely used 64-bit operating systems for Itanium processors are at this time Microsoft Windows[†] XP 64-bit Edition, Linux[†] in various flavors, and HP-UX 11i from Hewlett-Packard Company. The 64-bit C/C++ or FORTRAN compilers available for Windows environments currently come

from Intel Corporation and from Microsoft Corporation. For Linux, Intel and GNU compilers are available. For HP-UX, Hewlett-Packard Company developed corresponding compilers. Other operating systems, compilers, and software development tools for Itanium-based systems not mentioned here are also available or under development.

Hardware based on Itanium processors had already been in use for more than a year at the time this book was published. Servers and workstations with up to 16 Itanium 2 processors are or will become available in 2002 from several manufacturers.

Instruction-Level Parallelism and the Itanium® Architecture

The Itanium architecture was designed for high-performance computing. It is a 64-bit architecture, and incorporates several features that allow for exploitation of parallelism present in source code. The notion of instruction-level parallelism is presented next, followed by a discussion of the main performance limiters when trying to exploit it, along with methods used in the Itanium architecture to reduce the negative effects of these limiters.

Instruction-Level Parallelism

Parallelism exhibited by the instruction stream of a particular thread of execution is referred to as *instruction-level parallelism* (ILP). Exploiting ILP contributes directly to the speed of a given thread. ILP is to be distinguished from *thread-level parallelism* (TLP), where two or more separate threads of execution can exist in parallel. By using multiple CPU cores accessing the same memory system (symmetric multiprocessing, or SMP), thread-level parallelism can also be exploited. An intermediate alternative is symmetric multi-threading (SMT), where the core arithmetic-logic and floating-point units, as well as certain parts of the instruction execution logic are shared between multiple threads with separate dispatch mechanisms. While all these techniques can be useful in running multiple processes or a single process that naturally separates into multiple threads, they are largely orthogonal to the question of enhancing ILP and will not be considered further here.

The effect of an instruction stream is usually described using a serial model of how every instruction modifies the machine state. However, the effect of the program is often unchanged if certain groups of instructions are permuted or executed in parallel. In most cases, this is possible

when the output of any instruction is not an input or an output to one of the others. For example, the two-instruction sequence

```
x = y + 1;  
z = x + y;
```

cannot be modified, because the second instruction relies on the result of the first. On the other hand, the instructions in the sequence

```
x = y + 1;  
z = w + y;
```

can be reordered or executed in parallel (for distinct variables w , x , y , and z). In larger groups of instructions, there is often quite extensive potential for parallelism.

The Itanium architecture provides means that allow much of the available parallelism to be specified by software at compilation time, rather than requiring the processor to determine it dynamically using specialized hardware. A significant amount of information is thus passed by the compiler or programmer to the processor, which may not be feasible to reconstruct from object code in traditional architectures.

However, certain properties that would allow for greater parallelism can in general be determined only at run time. To help manage such determinations dynamically, the Itanium architecture provides, for example, features for establishing at run time whether two pointers are to overlapping regions of memory, or to avoid raising exceptions for code executed speculatively.

Some of the performance limiters that may prevent parallel execution of instructions and the ways in which the Itanium architecture works around them are presented next.

Itanium Architecture Solutions to Limitations on ILP

In practice, the ability of a processor to reorder instructions itself for parallel execution is limited. Often this is not because of an inherent required sequence in the original source program, but rather because the instruction sequence is forced to impose its own seriality owing to architectural limitations. Some of the major performance limiters when trying to execute instructions in parallel can be the limited number of registers, the memory latency, possible aliasing, branches, and non-obvious implicit parallelism. The Itanium architecture has several provisions to reduce the negative effects of these factors, as indicated by the following examples.

Limited Number of Registers

Consider the code sequence

```
x = y + 1;
z = y + x;
w = u + v;
```

The third instruction is independent of the other two, and can therefore be executed in parallel with the first. However, if the number of registers available is limited (in conjunction with other instructions not shown here), and the result of x is not used subsequently, the compiler may choose to reassign the same register for w as was used for x . The effective instruction stream is then

```
x = y + 1;
z = y + x;
x = u + v;
```

This effectively hides the potential for parallelism. Although sophisticated out-of-order execution and register renaming logic can detect and avoid many of these pseudo-dependencies, performance is still usually inferior to that obtained if the instructions were placed in the order

```
x = y + 1;
w = u + v;
z = y + x;
```

Similar problems arise from the use of two-address instructions where the result is written to the same register as was used for one of the arguments.

To allow the natural coding of parallelism, the Itanium architecture provides a large register set, with 128 general registers and 128 floating-point registers, and allows results of operations to be written back to an arbitrary destination register. Furthermore, a transparent method of register renaming and rotation supports very compact and efficient coding of procedure calls and of software-pipelined loops.

Memory Latency

Memory technology has not achieved the same performance gains as the processor technology, since processor clock frequencies increased at a much faster pace than memory bus clock frequencies. Thus, load access times not only from main memory but also from the different levels of cache memory have an increasingly negative impact on performance, when considered in relative terms. Store operations do not in general have the same effect, because the first instruction in many

dependency chains is often a load instruction, with the store occurring at the end. When several pipelined execution units are available, a processor will not be able to fully utilize them unless a sufficient number of instructions and data operands can be fetched from memory in every clock cycle. To alleviate the effects of memory latency, the Itanium architecture provides prefetch instructions for filling in cache lines before they are actually necessary. Also, prefetch hints are provided that can be attached to branch instructions, to make sure that code to be executed is readily available when needed. These hints determine prefetching of instructions along the path that is more likely to be taken following the branch instruction.

Aliasing

Reordering instructions depends on knowing which variables are distinct. However, it can be impossible to determine this at compilation time. For example, assuming that in the following assignments x and y are pointing at non-overlapping regions of memory and u and v are distinct, it is safe to reorder or parallelize the operations

```
*x = u + 1;
v = *y + 2;
```

The compiler may in common cases be able to determine this at compilation time, or even assume it because of programming language features, for example if the pointers x and y are of different types in C. However, the general problem of deciding this is unsolvable. In order to allow reordering in the common case where the regions do not overlap, the Itanium architecture features an *advanced load* instruction. A load is performed in advance in parallel with other operations, as if no overlapping occurred, thus helping to hide the load operation latency. A later check is performed at the point of use for overlap with an intervening write, and the result is corrected if such a write had occurred. This technique is referred to as data speculation.

Branches

Branches are a major hurdle when trying to find instructions to execute in parallel. Besides prefetch hints, other mechanisms are provided in the Itanium architecture to reduce the negative effects of branches on performance.

Eight branch registers support indirect branches and an efficient call-return mechanism. In addition, an advanced prediction logic ensures a reduced number of mispredicted branches.

Branch instructions followed by short alternate paths can be eliminated through predication in the Itanium architecture. Predication is the conditional execution of instructions, based on the value of a logical predicate associated with most instructions. If the predicate is true, the instruction is executed as expected. If it is false, the predicated instruction is reduced to a `nop` (no operation). This creates the opportunity for more instruction-level parallelism. A set of 64 predicate registers support this technique.

The last technique mentioned here is control speculation, which can be related both to branch instructions, and to reducing the effects of memory latency. In most processor architectures, instructions that follow a branch, including load operations, cannot be executed in advance in parallel with other instructions. The reason for this is that sometimes instructions do not update the processor state in the usual way, but instead generate exceptions or cause some interaction with the outside world or other components of the computer system. If such instructions are reordered, there can be a change in the behavior of the program. For example if a load is controlled by some condition, the failure of that condition may cause the load to generate an exception. Thus we cannot freely issue the load first, since it might generate the exception even when it wasn't supposed to be performed. Considering the code sequence

```
if (i >= 0)
    s += a[i];
else
    u = s;
```

The load cannot be moved above the test as in

```
t = a[i];
if (i >= 0)
    s += t;
else
    u = s;
```

because it may cause an exception if the condition `i >= 0` fails (reading outside the allocated region of memory).

To circumvent such situations, the Itanium architecture includes a *speculative load* instruction that tries to perform a load, but defers raising any exceptions it may cause. A later check instruction, executed only when the load is known to be necessary, will then propagate any deferred exceptions. Several other instructions in the Itanium architecture are speculative without being related to memory accesses, includ-

ing the computational floating-point instructions that write their results to floating-point registers.

Implicit Parallelism

The instruction-level parallelism in code compiled for CISC and RISC processors is implicit, and has to be uncovered by complicated hardware logic at run time, for out-of-order execution. The window in the instruction sequence where parallelism can be looked for is inherently limited in this case by the number of instructions that the processor can keep track of at a given time.

Parallelism is made explicit in the Itanium architecture, which has an instruction format where sequences of parallelizable operations can be grouped together in *instruction groups*, separated by *stop bits*. This means breaking away from the model where every instruction might depend on the previous one. The Itanium architecture instruction format, in conjunction with the array of features already mentioned, allows for the extraction of greater parallelism. For execution purposes, instructions are packed together in 128-bit *bundles* of three 41-bit instructions. The process of organizing instructions in bundles is independent of organizing the instructions into instruction groups. The larger instruction size when compared to traditional 32-bit instructions is caused by the larger register files available in Itanium processors, and by the uniform predication of instructions. These benefits outweigh the increase in code size, however. Five bits in each bundle contain a template identifier, specifying the execution units to be used and possibly any stop bits. The dispatch mechanism used ensures that code compiled for one implementation of the Itanium architecture can be executed on any subsequent implementation.

Summary of Itanium Architecture Features

In summary, the Itanium architecture is characterized by having a large number of registers, making use of explicit instruction-level parallelism that is not limited by a hardware window, being able to use techniques such as predication and speculation, and ensuring compatibility between successive implementations. The instruction set allows programmers and compilers to give the processor performance hints on how the code is expected to run. Floating-point performance is enhanced through several innovations. As designed, the Itanium architecture offers flexibility and scalability for several future implementations, and is poised to become the best choice for mid-range and high-end servers and workstations.

Scientific Computing

Satisfying the requirements imposed by scientific computing, based on floating-point computations, was from the beginning one of the more demanding aspects of processor and system design. The first microprocessors were used mainly as controllers in devices ranging from traffic lights to automated lab equipment. In many cases floating-point calculations were needed, but there was no corresponding hardware support yet. Instead, floating-point emulation libraries were used, implemented based on integer operations. Emulators continued to exist even when processors started being used in personal computers. Later, numeric coprocessors were added to perform floating-point computations more quickly. Eventually, as more complexity could be handled in hardware, floating-point units were incorporated in processors along with integer, memory, branch, and other execution units.

A serious issue in the 1970s and early 1980s was that various processors were performing floating-point computations according to slightly different rules. This problem was resolved through the adoption by virtually all commercial processor manufacturers of the IEEE Standard 754-1985 for Binary Floating-Point Arithmetic [2]¹. The standard defines basic and extended floating-point formats, basic floating-point operations and their properties, as well as floating-point exceptions and their handling. Most provisions presented in the standard are mandatory, but some are optional and may be included or not. Processors conforming to this standard can implement floating-point operations entirely in software, entirely in hardware, or in any combination of software and hardware. This is especially important, because even in the most modern processors of today some floating-point operations are considered too complex to be implemented in hardware for their added benefit, and are rather left to software, implemented in microcode, in the operating system kernel, or at user application level.

Floating-point hardware followed the general trends of processor evolution. Multiple execution units and pipelined execution units increased the floating-point throughput for technical and scientific computations. Advanced numerical algorithms reduced the latency of the more complex floating-point operations. New capabilities were added to floating-

¹ This standard was followed by IEEE Standard 854-1987 for Radix-Independent Floating-Point Arithmetic. All subsequent references to the IEEE Standard are to the IEEE Standard 754-1985 for Binary Floating-Point Arithmetic [2].

point execution units to make computations more reliable, stable, and fast.

The Itanium floating-point architecture was developed along the same lines. Its high performance, accuracy, and flexibility characteristics make it ideal for technical computing. Floating-point enhancements include a high precision and wide range basic floating-point data type, the fused floating-point multiply-add operation, software division and square root operations, and a large number of floating-point registers. Floating-point code can also draw on other generic Itanium architecture features such as predication, register rotation, high memory bandwidth, and speculation.

All floating-point data types are mapped internally to an 82-bit format, with 64 bits of accuracy and a 17-bit exponent. This affords calculations that are more accurate, and do not underflow or overflow as often as on other processors. The great flexibility in using and combining various floating-point formats and computation models makes it easy to implement complex numerical algorithms more efficiently than before.

The fused multiply-add operation combines two basic floating-point operations in one, with only one rounding error. Besides the increased accuracy, this can effectively double the execution rate of certain floating-point calculations, as the fused multiply-add operation forms an efficient computation core that maps perfectly to several common algorithms used for technical and scientific purposes.

The fused multiply-add operation creates the possibility of implementing new algorithms, such as software-based division and square root operations. As execution units are pipelined, a division or square root operation does not block the floating-point unit for the entire duration of the computation, and several other operations can be initiated or carried out in parallel.

The large number of floating-point registers available, of which some are static and some are rotating, allows for efficient implementation of complicated floating-point calculations. An illustration of software and hardware interaction in the Itanium architecture, this is achieved on one side by avoiding frequent accesses to memory, and on the other through software pipelining of loops containing floating-point computations. For example, the throughput for division operations can be as high as one result for every four clock cycles on the Itanium and Itanium 2 processors.

Organization of this Book

As stated, the Itanium architecture is best suited for applications that require large amounts of memory, high bandwidth and processing power, and/or intensive floating-point calculations. The present text focuses on those aspects of the architecture that make it perform well in scientific and engineering applications. For completeness though, other general features of the architecture are also presented in the first part of the book. Whenever possible in such cases, the floating-point aspects are stressed and illustrated.

This text is intended to be a useful learning tool for computer scientists and application programmers in the scientific and engineering fields, but can also be used by compiler and operating system writers, students who want to learn about this new processor architecture, and in general by anyone writing, porting, or using floating-point code on Itanium processors.

Chapter 2 presents the processor architecture, as seen by the application programmer. The application registers, the register stack used in procedure call/return, predication, branching, register rotation, and modulo-scheduled loop support are covered.

Chapter 3 focuses on the Itanium processor floating-point architecture, and presents in more detail aspects that were already mentioned in passing in Chapter 1.

Chapter 4 presents the topics of memory access, control speculation, and data speculation. Speculation, often related to memory accesses, is most likely to be implemented by compilers. Therefore high-level as well as assembly language application writers will primarily be interested in understanding how it works, but will not apply it directly in most cases.

Chapter 5 contains an introduction to Itanium processor assembly language programming, and also a description of the software conventions that make various software modules generated by different tools or in various environments work together. Information from this chapter is useful for writing assembly code, or for understanding assembly language code generated by compilers.

Chapter 6 describes microarchitecture characteristics of the Itanium and Itanium 2 processors. The core pipeline, functional units, issue and dispersal rules, and the memory hierarchy are described, allowing the reader to understand some of the inner workings of the processor that have a direct effect on performance.

Chapter 7 is dedicated to the subject of floating-point exception handling. A more specialized and usually less understood topic, this will be of interest to those seeking more in-depth knowledge of the Itanium floating-point unit. In particular, scientific and engineering application code writers should be aware of the existence of floating-point software assistance requests in Itanium and Itanium 2 processors.

Chapter 8 presents the software implementations of the floating-point division, square root, and remainder operations. Besides algorithms that implement these operations, some mathematical background is provided that shows how the underlying floating-point architecture can be used for designing efficient and accurate algorithms. Integer division and remainder are also presented in this chapter as their implementation is realized in software, based on floating-point operations.

Chapter 9 is a collection of basic examples considered interesting and representative as floating-point applications: polynomial evaluation, complex arithmetic, quad precision arithmetic, applications of special instructions, and software pipelining examples. It should be noted that software pipelining is introduced in Chapter 2, where register rotation and modulo-scheduled loop support are described. (Chapter 5 also contains a few examples of software-pipelined loops that are used to illustrate microarchitectural features.)

Chapter 10 presents the most important mathematical functions available to users at run time, together with performance and accuracy characteristics. It also presents the floating-point compilation model designed to make floating-point computations more predictable and efficient across all platforms based on the Itanium architecture.

Chapter 11 presents a number of selected kernel computations, representative for scientific and engineering applications: interval and high precision arithmetic, Fast Fourier Transforms, basic linear algebra routines and libraries, vector mathematical libraries, and cryptography kernels.

Readers might not be interested equally in all the chapters of this book. For a general understanding of how floating-point works in the Itanium architecture, Chapters 1 through 4 will be sufficient. Scientific and engineering code programmers will continue with Chapter 5, and will also find part of Chapter 7, and Chapters 9 and 10 interesting. Chapter 6 is needed by those who wish to achieve maximum performance with their computation kernels. Chapter 8 is useful for understanding how division and square root work. Numerical algorithm designers will find

it, as well as Chapter 7, interesting. Chapter 11 will be read by scientific and engineering code programmers who are using platforms based on Itanium processors for solving technical problems, or want to become experts in this field.

