Intel® Itanium® Floating-Point Architecture

Marius Cornea, John Harrison, and Ping Tak Peter Tang

Intel Corporation

Abstract

The Intel® Itanium® architecture is increasingly becoming one of the major processor architectures present in the market today. Launched in 2001, the Intel Itanium processor was followed in 2002 by the Itanium 2 processor, with increased integer and floating-point performance. Measured by the SPEC CINT2000 benchmarks, the Itanium 2 processor still trails by about 25% the Intel P4 processor in integer performance, albeit P4 runs at more than three times Itanium's clock frequency. However, its floating-point performance clearly leads in the SPEC CFP2000 charts, and its rating is about 25% higher than that of the P4 processor. While the general features of the Itanium architecture such as large register sets, predication, speculation, and support for explicit parallelism [1] have been presented in several papers, books, and mainstream college textbooks [2], its floating-point architecture has been less publicized. Two books, [3] and [4], cover well this area. The present paper focuses on the floating-point architecture of the Itanium processor family, and points out a few remarkable features suitable to be the focus of a lecture, lab session, or project in a computer architecture class.

Introduction

The performance of today's 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 [5], the Itanium architecture was co-developed by Intel Corporation and Hewlett-Packard Company, combining the best in the RISC and VLIW architectures, while also adding several features originating 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.

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, increased the performance level of the Itanium processor by a factor of 1.5 to 2 in several cases.

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.

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. Itanium processor features that benefit the latter category will be the focus of the present paper.

Scientific and engineering applications that can take advantage of the increased floating-point performance of Itanium processors 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.

Itanium Floating-Point Architecture

The Itanium floating-point architecture has been designed to combine high performance and good accuracy. A large floating-point register set of 128 registers is provided, and almost all operations can read their arguments from, and write their results to,

arbitrary registers. Together with register rotation for software-pipelined loops, this large number of registers allows the encoding of common algorithms without running short of registers or needing to move data between them in elaborate ways. Registers can store floating-point numbers in a variety of formats, and the rounding of results is determined by a flexible combination of several selectable defaults and additional instruction completers.

The basic arithmetic operation, the *floating-point multiply-add* (*fused multiply-add*), allows higher accuracy and performance in many common algorithms. Several additional features are also present to support common programming idioms. 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 3.5 clock cycles on the Itanium and Itanium 2 processors.

Floating-Point Formats

The IEEE Standard 754-1985 for Binary Floating-Point Arithmetic [6] mandates precisely defined floating-point formats referred to as single precision and double precision. As well as these IEEE-mandated formats, Intel architectures have traditionally supported a double-extended precision type, with 64 bits of precision and a 15-bit exponent field. In current IA-32 implementations, results computed in the floating-point register stack may be rounded to 24, 53 or 64 bits of precision. Although the first two precisions coincide with the IEEE single and double precision, the precision control setting in IA-32 processors does not affect the exponent range, as the exponent uses a 15-bit field until the number is actually written back to memory. Although the greater exponent range is normally advantageous, it can lead to subtle variations in underflow and overflow behavior depending on exactly when a result is written to memory (which may be compiler-dependent and hard to predict).

In order to maintain the useful extra exponent range but allow the user complete control over rounding, the Itanium architecture allows for *both* conventional single and double precision formats *and* formats with the same precision but a 15-bit exponent field. In addition, a still wider exponent field of 17 bits is provided in each case, a very useful feature for intermediate calculations with double-extended precision numbers. This means that there are actually eight floating-point formats directly supported by the Itanium architecture, shown in Table 3-1.

Table 3.1. Floating-Point Formats Available in the Itanium Architecture

Format	Precision	Exponent Bits	Exponent Range
Single	24	8	–126 to 127
Double	53	11	–1022 to 1023
Double extended	64	15	–16382 to 16383
IA-32 stack single	24	15	–16382 to 16383
IA-32 stack double	53	15	–16382 to 16383
Register single	24	17	–65534 to 65535
Register double	53	17	–65534 to 65535
Register	64	17	–65534 to 65535

Register and Memory Encodings

The Itanium architecture specifies 128 floating-point registers f0, f1, ..., f127. Register f0 is hardwired to +0.0 and f1 to +1.0, and both are read-only, but all other registers are available for reading and writing. Each register is 82 bits long, with a 64-bit significand (using an explicit integer bit), a 17-bit exponent field and a 1-bit sign. The exponent bias has the value 65535, or 0xFFFF (hexadecimal).

Certain values, such as NaNs, are neither negative nor positive. Special encodings, such as zeros, infinities, pseudo-zeros, pseudo-denormals, NaNs, pseudo-NaNs, pseudo-infinities, or NaTVal are all possible. Some of these special categories are explained below.

The minimum (biased) exponent value of 0 is reserved for double-extended real denormalized numbers (*denormals*), and for *pseudo-denormals*. The maximum (biased) exponent value of 131071, or 0x1FFFF, is reserved for special numbers such as infinities and NaNs.

Other exponent values, between 0 and 0x1FFFE in biased form, are used for finite numbers. The value in a floating-point register with sign *s*, biased exponent *e* and significand $m_0m_1m_{2...}m_{63}$ is determined by the following formula for biased 17-bit exponents between 1 and 0x1FFFE:

$$(-1)^s \cdot 2^{e-65535} \cdot m_0.m_1m_{2...}m_{63}$$

and the following for biased exponents that are zero:

$$(-1)^s \cdot 2^{-16382} \cdot m_0.m_1m_{2...}m_{63}$$

The register encoding is redundant: the same real value can sometimes be represented in several different ways. This is a consequence of the presence of an explicit integer bit, and is true of all floating-point formats that support it. For example, one can have positive *pseudo-zeros* with significand equal to zero but exponent from 0x000001 to 0x1FFFD rather than zero. Most of these alternative representations of the same number are equally acceptable as inputs to floating-point operations, the only exceptions being the *unsupported* numbers with exponent 0x1FFFF and integer bit 0 (pseudo-infinities and pseudo-NaNs). In particular, the user can freely operate on arguments of mixed format without any time-consuming format conversions. This is often useful, especially when:

- Using double-extended intermediate precision calculations to compute a double precision function. The double precision input argument can be freely combined with double-extended intermediate results.

- Computing functions involving constants with few significant digits. Whatever the precision of the computation, the short constants can be stored in single precision.

However, *results* of floating-point operations, and floating-point values loaded from memory, are always mapped to fixed canonical representatives in the register.

Note that the subsets of positive and negative register format numbers are almost symmetrical, with only two exceptions. First, NaTVal, the special *Not a Thing Value* quantity used to track floating-point computations that encounter failed speculative loads, has an encoding as an otherwise unused positive floating-point number: positive sign, biased exponent of 0x1FFFE and significand of 0 (a pseudo-zero). Second, encodings with a positive sign and a biased exponent of 0x1003E (corresponding to the unbiased decimal value of 64) are used also for *canonical integers*, and for *SIMD[1] floating-point numbers* (pairs of 32-bit single precision numbers). These are stored in the significand portion of a floating-point register.

The register encoding used differs from the encoding used when floating-point values are stored in memory. Single precision and double precision floating-point numbers are stored in the memory format specified by the IEEE Standard, with exponent biases of 127 (0x7F) and 1023 (0x3FF) respectively, and no explicit integer bit. Double-extended and register format numbers are stored in a more direct mapping of the register contents (the exponent bias for double-extended values is 0x3FFF).

For example, the value of a single precision floating-point number with sign *s*, biased exponent *e* and significand $m_0m_1m_{2...}m_{23}$ stored in memory is determined by the following formula for biased 8-bit exponents between 0x1 and 0xFE:

$$(-1)^s \cdot 2^{e-127} \cdot m_0.m_1m_{2...}m_{23}$$

and the following for biased exponents that are zero:

$$(-1)^s \cdot 2^{-126} \cdot 0.m_1m_{2...}m_{23}$$

For double precision values, the exponent bias to subtract from the exponent *e* is 1023, and denormals have an exponent of –1022. For double-extended precision values, the exponent bias is 16383, and denormals have an exponent of –16382.

Status Fields and Exceptions

Given the number of floating-point formats available in the Itanium architecture, it is important to have a flexible means of specifying the desired floating-point format for a particular result to be rounded into, as well as the direction of rounding (e.g. rounding to nearest or truncation). Moreover, in accordance with the IEEE Standard, floating-point operations on the Itanium architecture not only produce results, but may optionally trigger exceptions or record exceptional conditions by setting sticky status flags. It would be impractical to encode all this information into the

[1]SIMD is an acronym for Single Instruction and Multiple Data, a form of parallel computing in which one operation is performed in parallel on multiple sets of operands.

format of each instruction, so some global status and control word is necessary for specifying options as well as recording exception flags. On the other hand, having only a single record would be inconvenient where there are several parallel threads of control, or where exceptions in some intermediate instructions need to be ignored. Therefore, the Itanium architecture features four different s*tatus fields* which can be specified by completers in most floating-point instructions. An instruction with a given status field completer is then controlled by, and records certain information in, that status field.

A 64-bit *Floating-Point Status Register* (FPSR) controls floating-point operations and records exceptions that occur. The FPSR contains 6 *trap disable* bits that control which floating-point exception conditions actually result in a trapped exception (where control passes to the OS and possibly to a user handler), and which are merely recorded in sticky status flags. These bits control the five IEEE Standard exceptions: invalid operation (vd), division by zero (zd), overflow (od), underflow (ud) and inexact result (id), as well as the additional *denormal/unnormal operand* exception (dd), which occurs if an input to a floating-point instruction is an unnormalized number. In addition to this field, the FPSR contains four 13-bit status fields, denoted in the assembly language syntax by s0, s1, s2 and s3.

Each status field can be divided into two parts: *flags* and *controls*. The six flags are bits that record the occurrence of each of the 6 exceptions mentioned above, when exceptions are masked, or, for the overflow, underflow or inexact result exceptions, also when they are *enabled* (*unmasked*). These flags are *sticky*, meaning that later operations that do not cause exceptions will not set flags back to 0, so the occurrence of an exception anywhere in a computation sequence will be apparent at the end of that sequence. Of the control part, one bit (td) allows all exceptions to be disabled irrespective of the individual trap disable bits from the FPSR (often useful in intermediate calculations). The remaining 6 bits control the rounding mode, precision and exponent width, and the flushing to zero of tiny[2] results.

[2]The IEEE Standard allows for two methods of determining whether a result is tiny. Intel architecture processors choose to define a result as being tiny if the exact value rounded to the destination precision *while assuming an unbounded exponent* is less than the smallest normal value that can be represented in the given floating-point format.

The pc and wre fields together determine the floating-point format into which the result will normally be rounded. The rounding control rc determines the IEEE rounding mode.

Although the status fields determine the default rounding behavior of operations, it is often possible to override them by explicit completers. This applies, for example, to many of the instructions to be discussed below. If an instruction has an explicit .s or .d completer, then the destination format is single or double precision respectively, except if the wre flag is set, in which case register single or register double is used.

Software conventions for the FPSR determine many of the appropriate applications for particular status fields. Typically, s0 is the main user status field used for most floating-point calculations. Status field s1, with wre enabled and all exceptions disabled, is used for intermediate calculations in many standard numerical software kernels such as those for division, square root, and transcendental functions. Status fields s2 and s3 are also commonly used for speculation. The default setting of the FPSR is such that all status fields use the 64-bit precision, the round-to-nearest rounding mode, and have floating-point exceptions and the flush-to-zero mode disabled. Only status field s1 uses the widest-range exponent.

The Floating-Point Multiply-Add

In most existing computer architectures, there are separate instructions for floating-point multiplication and floating-point addition. In the Itanium architecture, these are subsumed by a more general instruction, the *floating-point multiply-add* or *fused multiply-accumulate*, which takes three arguments, multiplies two of them and adds in the third. The basic assembly syntax is:

(*qp*) fma.*pc*.*sf* $f_1 = f_3, f_4, f_2$

which sets $f_1 = f_3 \cdot f_4 + f_2$. Note that no intermediate rounding is performed on the result of the multiplication, and the result is written to f_1 as if it were first computed exactly and then rounded, in a natural extension of the way conventional arithmetic operations are specified to behave in the IEEE Standard. The rounding of the result and the triggering of exceptions is controlled by the status field specified by the *sf* completer and possibly by the FPSR trap disable bits, except that the rounding precision from *sf* may be overridden by an optional precision control completer *pc*.

Since the floating-point registers f0 and f1 are hardwired to the values +0.0 and +1.0 respectively, addition and multiplication can easily be implemented

as special cases of the fma: $x + y = x \cdot 1 + y$ and $x \cdot y = x \cdot y + 0$. In fact, the floating-point addition and multiplication assembly instructions

(*qp*) fadd.*pc*.*sf* $f_1 = f_3, f_2$
(*qp*) fmpy.*pc*.*sf* $f_1 = f_3, f_4$

are simply pseudo-operations that expand into

(*qp*) fma.*pc*.*sf* $f_1 = f_3$, f1, f_2
(*qp*) fma.*pc*.*sf* $f_1 = f_3, f_4$, f0

respectively. In order to change signs, there are two variants of the fma: the fms (floating-point multiply-subtract) and fnma (floating-point negative multiply-add). The instructions

(*qp*) fms.*pc*.*sf* $f_1 = f_3, f_4, f_2$
(*qp*) fnma.*pc*.*sf* $f_1 = f_3, f_4, f_2$

compute $f_1 = f_3 \cdot f_4 - f_2$ and $f_1 = -f_3 \cdot f_4 + f_2$ respectively. Floating-point subtraction

(*qp*) fsub.*pc*.*sf* $f_1 = f_3, f_2$

is likewise a pseudo-operation for

(*qp*) fms.*pc*.*sf* $f_1 = f_3$, f1, f_2

An even more degenerate instance of fma, called fnorm (floating-point normalize) can be used to round a result into a given floating-point format. This can be used as a 'lowering' operation to convert a value to a smaller floating-point format, but the most common use is just to ensure that the number is normalized. (This is often useful, because processing unnormalized values is slower in most cases than performing an fnorm followed by the intended operation.) This rounding to a canonical value is accomplished by the standard fma behavior, and so fnorm.*pc*.*sf* $f_1 = f_3$ is simply a pseudo-operation for fma.*pc*.*sf* $f_1 = f_3$,f1,f0.

It was stated above that the fma behaves in accordance with the IEEE Standard. Strictly speaking, that standard does not cover the fma instruction, but all the stipulations are extended to it in a natural way. However, there is some subtlety over the signs of zero results.

If the result of an fma without the final rounding would be nonzero, then should it round to zero, the sign of the final zero will reflect the sign of the exact result. This of course is the 'correct' decision, but is a non-trivial extrapolation of the IEEE Standard. Here, the sign rules for multiplications and divisions are obvious (the exclusive or of the input signs). And for addition and subtraction, when the rounded result is nonzero, the exact result must be too (in a fixed floating-point format), so only the special case of exactly zero results needs to be dealt with.

Now consider the case when the result of an fma instruction without rounding is exactly zero. Normally, the sign of $x \cdot y + z$ is determined by multiplying the signs of x and y to give a sign for the intermediate result, then using the rules of the IEEE Standard, treating $w + z$ as if it were an ordinary sum. However, this is not appropriate for considering the ordinary product a special case of the fma. For example, $(+1.0) \cdot (-0.0) + (+0.0)$ would give +0.0, whereas the IEEE-specified product is (–0.0). This difficulty is circumvented as follows: if the third argument to the fma is actually register zero (f0), then the sign of zero is determined by the IEEE rules for products. Otherwise, the sign of zero results is decided as specified above for fma, even if the third argument to fma is not the special register zero f0 but nevertheless contains the value zero. This applies equally to the variants fms and fnma.

A floating-point multiply-add is a very valuable architectural feature, for reasons of both speed and accuracy. In typical implementations, the final addition can be combined into the floating-point multiplication operation without significantly increasing its latency. Thus, a single fma is faster than a multiplication and an addition executed successively. Since additions and multiplications are heavily interleaved in many important floating-point kernels (the evaluation of inner, or dot, products of vectors or the evaluation of polynomials for example), the use of an fma can lead to significant performance improvements. For example the vector dot product $x \cdot y$:

$$p = \sum_{i=0}^{N-1} x_i \cdot y_i$$

can be evaluated by a succession of fma operations of the form

$$p = p + x_i \cdot y_i$$

requiring only n floating-point operations, whereas with a separate multiplication and addition it would require $2n$ operations, with a longer overall latency.

Apart from its speed advantage, the fact that no intermediate rounding is performed on the product also tends to reduce overall rounding errors. In common cases this difference may be relatively unimportant, but in special situations, the lack of an intermediate rounding makes possible a number of techniques that are difficult or costly on a traditional architecture. The floating-point division and square root implementations provide ample illustration of this fact, but here are three other characteristic examples.

Exact Arithmetic

In certain applications it is important to perform arithmetic to very high precisions, perhaps hundreds of bits. A natural way of manipulating very precise

numbers is as *floating-point expansions*; that is, sums of standard floating-point numbers of decreasing magnitude. In order to perform efficient computations on such expansions, the basic building blocks are operations that compute exact arithmetic operations on individual pairs of floating-point numbers. For example, it is known (Moller [7], and Dekker [8]) that if $|x| \geq |y|$ the exact sum $x + y$ can be obtained as a 2-piece expansion Hi + Lo by the following sequence of floating-point adds:

$$Hi = x + y$$
$$tmp = x - Hi$$
$$Lo = tmp + y$$

This is straightforward to implement on traditional architectures, though features of the Itanium architecture make it significantly more efficient. However, on traditional architectures there is no similarly easy way of obtaining the exact product of floating-point numbers as an expansion; this requires fairly complicated and inefficient methods based on splitting the numbers into high and low parts by masking and performing numerous sub-computations. However, with the fms instruction, this computation is simple and efficient:

$$Hi = x \cdot y$$
$$Lo = x \cdot y - Hi$$

This sequence always results in $Hi + Lo = x \cdot y$ exactly with Lo a rounding error in $Hi \approx x \cdot y$.

Accurate Remainders

It is often the case that given a floating-point number q approximately equal to the quotient a / b of two floating-point numbers, one wants to know the remainder $r = a - b \cdot q$. This arises whenever evaluation of a quotient to higher precision is needed, for example, in floating-point expansions. Provided the approximation q is good enough, it can be shown that r is always representable exactly as a floating-point number. However, that does not mean it is always straightforward to obtain it on traditional architectures. In fact, if $a - b \cdot q$ is computed by a multiplication and a subsequent subtraction, the rounding error in the multiplication may be comparable in size to r itself, rendering the result essentially meaningless. Thus, complicated masking and multiple computations are necessary. But in the Itanium architecture, evaluating $a - b \cdot q$ by an fnma instruction will give an exact answer provided q is accurate enough.[3]

Accurate Range Reduction

A similar situation arises when one has an integer approximation to the exact quotient. Many algorithms for mathematical functions, in particular trigonometric functions such as *sin*, begin with an initial range reduction phase, subtracting an integer multiple of a constant such as $\pi / 2$. With the fma this can be done in a single instruction $x - N \cdot P$ yielding an accurate result. Without the fma however, the rounding error in the multiplication could severely distort the result, so it might be necessary to represent P as the sum of two numbers with fewer significant bits. (Each of these numbers can be multiplied by N without error, and after several operations the main result can be obtained.) The fma is also useful for obtaining the appropriate N rapidly in the first place. Typically, one wants to perform some operation such as

$$y = Q \cdot x$$
$$N = rint\ (y)$$
$$r = x - N \cdot P$$

where *rint* (y) denotes the rounding of y to an integer, and $Q \approx 1 / P$. Rather than using the special fcvt instructions to convert y to an integer, the integer conversion can be performed by adding and subtracting a large constant like $S = 2^{p-1} + 2^{p-2}$ where p is the floating-point precision, for example $p = 53$ for double precision. (Adding such a constant fixes the most significant bit of the sum and hence performs integer rounding of y, provided $|y| \leq 2^{p-2}$; the use of 2^{p-2} makes the technique work for both positive and negative y.) Using the fma the multiplication by Q and the addition of S can be combined, and hence the reduced argument can be obtained by just three fma operations:

$$y = S + Q \cdot x$$
$$N = y - S$$
$$r = x - N \cdot P$$

This approach has the additional advantage of avoiding some rare problems with the intermediate rounding of the product $Q \cdot x$.

Comparison and Classification

Floating-point comparisons are similar to the integer comparisons. The basic instruction is

---

[3] It suffices for q to be accurate to one *unit in the last place* (ulp).

(*qp*) fcmp.*fcrel.fctype* $p_1, p_2 = f_2, f_3$

Here the *fcrel* completer, which is compulsory, determines the relation that is tested for. The mnemonics differ slightly from those used in the integer comparison: eq for $f_2 = f_3$, lt for $f_2 < f_3$, le for $f_2 \le f_3$, gt for $f_2 > f_3$, ge for $f_2 \ge f_3$,and unord for $f_2 ? f_3$. There is no signed/unsigned distinction but there is a new possibility, shown in the last case ($f_2 ? f_3$): two values may be *unordered*, since a NaN (Not a Number) compares false with any floating-point value, even with itself. Mnemonics are also provided for the complements of all these conditions, although in the actual instruction encoding these simply swap the predicate registers and/or the input floating-point registers.

The *fctype* field has two possible values, *none* (i.e. the field is omitted in the assembly syntax), and unc. If omitted, the result of the comparison and its complement are written to the designated predicate registers in the usual way. If the completer unc is used, however, then the behavior is the same if the qualifying predicate *qp* of the instruction is true, but *both* the predicate registers p_1 and p_2 are cleared if *qp* is false.

It is often desirable to classify a floating-point number, for example to abort a calculation if an input is infinite or NaN. A comprehensive instruction for classifying the floating-point value in a register is fclass:

(*qp*) fclass.*fcrel.fctype* $p_1, p_2 = f_2,$ *fclass*

The result of classifying the contents of f_2 is written to the predicate registers p_1 and p_2, controlled by the optional *fctype* in the same way as for comparisons (i.e. its values can be *none* or unc). The *fcrel* field may be m (f_2 is a member of the class specified by *fclass*) or nm (f_2 is not a member of the class specified by *fclass*). The actual classification is encoded as a 9-bit field whose bits are interpreted to determine whether the floating-point value is: positive or negative; zero, unnormalized, normalized or infinity; NaN or NaTVal.

Division and Square Root

There are no instructions specified in the Itanium architecture (except in IA-32 compatibility mode) for performing floating-point division or square root operations. Instead, the only instruction specifically intended to support division is the *floating-point reciprocal approximation* instruction, frcpa, which given floating-point numbers *a* and *b*, normally returns an approximation to 1 / *b* good to about 8 bits. The syntax of this instruction is as follows:

(*qp*) frcpa.*sf* $f_1, p_2 = f_2, f_3$

Similarly, the only instruction to support square root is the *floating-point reciprocal square root approximation* instruction frsqrta, which given a floating-point number *a*, normally returns an approximation to 1 / √a good to about 8 bits.

(*qp*) frsqrta.*sf* $f_1, p_2 = f_3$

In special cases such as $b = 0$ for frcpa or $a = 0$ for frsqrta, these instructions actually return the full IEEE-correct result for the relevant operation (the full quotient in the case of frcpa), and indicate this by clearing the output predicate register p_2. Usually, however, the initial approximations need to be refined to perfectly rounded quotients or square roots by software, and this is indicated by setting the predicate register p_2. Consequently, one can simply predicate the software responsible for refining the initial approximation by this predicate register. Thanks to the presence of the fma instruction, quite short straight-line sequences of code suffice to do this correction. There are several reasons for relegating division and square root to software.

- By implementing division and square root in software, they immediately inherit the high degree of pipelining in the basic fma operations. Even though these operations take several clock cycles, new ones can be started while others are in progress. Hence, many division or square root operations can proceed in parallel, leading to much higher throughput than is the case with typical hardware implementations.

- Greater flexibility is afforded because alternative algorithms can be substituted where it is advantageous. It is often the case that in a particular context a faster algorithm suffices, for example because the ambient IEEE rounding mode is known at compile time, or even because only a moderately accurate result is required (this might arise in some graphics applications).

- In typical applications, division is not an extremely frequent operation, and so it may be that the die area on the chip would be better devoted to something else.

Intel Corporation distributes a number of recommended algorithms for various precisions and performance constraints, so the user will not ordinarily have to be concerned with the details of how to implement these operations. As an example, consider the single precision division algorithm, optimized for throughput (it has the smallest possible number of floating-point instructions, resulting in the minimum latency per result in software-pipelined loops): The algorithm calculates $q = a/b$ in single precision, where *a* and *b* are single precision numbers, *rn* is the IEEE round to nearest mode, and *rnd* is any IEEE rounding mode. All other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

(1) $y_0 = 1 / b \cdot (1+\varepsilon_0)$, $|\varepsilon_0| < 2^{-8.886}$ table lookup
(2) $d = (1 - b \cdot y_0)_{rn}$ 82-bit register format precision
(3) $e = (d + d \cdot d)_{rn}$ 82-bit register format precision
(4) $y_1 = (y_0 + e \cdot y_0)_{rn}$ 82-bit register format precision
(5) $q_1 = (a \cdot y_1)_{rn}$ 17-bit exponent, 24-bit mantissa
(6) $r = (a - b \cdot q_1)_{rn}$ 82-bit register format precision
(7) $q = (q_1 + r \cdot y_1)_{rnd}$ single precision (IEEE)

The assembly language implementation follows [9], assuming the input values are in floating-point registers f6 and f7, and the result in f8:

```
      frcpa.s0 f8,p6=f6,f7;; // Step (1)  y0=1/b in f8
(p6) fnma.s1 f9=f7,f8,f1;;  // Step (2) d = 1-b*y0 in f9
(p6) fma.s1 f9=f9,f9,f9;;     // Step (3) e = d+d*d in f9
(p6) fma.s1 f8=f9,f8,f8;;  // Step (4) y1 = y0+e*y0 in f8
(p6) fma.s.s1 f9=f6,f8,f0;;   // Step (5) q1 = a*y1 in f9
(p6) fnma.s1 f6=f7,f9,f6;;   // Step (6) r = a-b*q1 in f6
(p6) fma.s.s0 f8=f6,f8,f9;; // Step (7) q = q1+r*y1 in f8
```

Support for software pipelining on Itanium processors allows for this algorithm to be scheduled without any additional code, so that one result is generated every 3.5 clock cycles (since there are 7 floating-point instructions to schedule on 2 floating-point units on Itanium and Itanium 2 processors). This is a lot more efficient than on most present-day processor architectures.

Table 3.2 shows the Itanium 2 processor cycle times for the division root algorithms of various precisions (a similar table is available for square root [9]). For algorithms optimized for latency, the operation latency is given, in number of clock cycles. For operations optimized for throughput, the number of clock cycles required to generate one result is given.

Table 3.2. Latency and Throughput for Floating-Point Division on the Itanium 2 Processor

Division	Single Precision	Double Precision	Double-Extended Precision
Optimized Latency	24	28	32
Optimized Throughput	3.5	5	7

The square root algorithms rely on loading constants, and the time taken to load these constants is not included in the overall latencies. If the function is inlined by an optimizing compiler, these loads should be issued early as part of normal operation reordering. For comparison, note that on the Itanium 2 processor, a floating-point add/subtract, multiply, or fused multiply-add operation has a latency of 4 clock cycles, and a throughput of 0.5 clock cycles (meaning that two results can be generated every clock cycle, for example in a software-pipelined loop).

Additional Features

The Itanium architecture includes a number of other useful floating-point instructions that have not been mentioned, which are covered in detail in [4]. They include:

- transferring values between floating-point and integer registers by means of the getf and setf instructions
- floating-point merging, useful in order to combine fields of multiple floating-point numbers to give a new number using the fmerge instruction
- floating-point to integer and integer to floating-point conversion using the fcvt instructions
- integer multiplication and division - the Itanium architecture does not specify a full-length integer multiplication or division instruction; instead, such operations are intended to be implemented using the floating-point unit, by first transferring the arguments to floating-point registers, performing the multiplication or division there, and transferring the result back
- floating-point maximum and minimum, using the fmax and fmin instructions

Conclusion

The Itanium floating-point architecture was designed so that 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 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.

The highest SPEC CFP2000 score for a single processor system, of 1431, belongs currently to an Itanium 2 system running at 1GHz - the Hewlett-Packard HP Server RX2600. The best performing P4 system, running at 3.06 GHz, has a score of 1092. The SPEC CINT2000 scores are in reverse order though – 810 and 1099 respectively. This gap will likely decrease and the advantage is expected to be on the Itanium processor family side as its core frequencies will get higher - today's Itanium processors run at relatively low frequencies, and as the compiler technology on which Itanium processors depend so much continues to evolve.

References

[1] Intel(R) Itanium(TM) Architecture Software Developer's Manual, Revision 2.0, Vol 1-4, Intel Corporation, December 2001

[2] John Hennessy, David Patterson, "Computer Architecture - A Quantitative Approach", Morgan Kauffman Publishers, Inc., third edition, 2002

[3] Peter Markstein, ''IA-64 and Elementary Functions: Speed and Precision", Hewlett-Packard/Prentice-Hall 2000

[4] Marius Cornea, John Harrison, Ping Tak Peter Tang, "Scientific Computing on Itanium-based Systems", Intel Press 2002

[5] John Crawford, Jerry Huck, "Motivations and Design Approach for the IA-64 64-Bit Instruction Set Architecture", Oct. 1997, San Jose, http://www.intel.com/pressroom/archive/speeches/mpf1097c.htm

[6] ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic, IEEE, New York, 1985

[7] O. Moller, "Quasi double-precision in floating-point addition", BIT journal, Vol. 5, 1965, pages 37-50

[8] T. J. Dekker, "A Floating-Point Technique for Extending the Available Precision", Numerical Mathematics journal, Vol. 18, 1971, pages 224-242

[9] "Divide, Square Root, and Remainder Algorithms for the Itanium Architecture", Intel Corporation, Nov. 2000, http://www.intel.com/software/products/opensource/libraries/numnote2.htm, http://developer.intel.com/software/products/opensource/libraries/numdown2.htm