Photomicrograph by Michael W. Davidson

Computational Aspects of the Pentium Affair

Tim Coe, Vitesse Semiconductor Corporation
Terje Mathisen, Norsk Hydro
Cleve Moler, The MathWorks
Vaughan Pratt, Stanford University

This is the story of exactly how the Pentium floating-point division problem was discovered, and what you need to know about the mathematics and computer engineering involved before deciding whether to replace the chip, install the workaround provided here, or do nothing.

THE PENTIUM AFFAIR HAS BEEN WIDELY PUBLICIZED. It started with an obscure defect in the floating-point unit of Intel Corporation's flagship Pentium microprocessor. It ultimately led to a deluge of traffic on the Internet, to hundreds of reports in the mass media, to bad jokes on late-night television, to a $475 million fourth-quarter debit on Intel's balance sheet, and to a surge in sales of jewelry made from recycled silicon.

Technically, what was behind it all? How did the world learn about the defect? How bad is it? How likely is it that scientific and engineering computations on Pentium-based machines will be affected? If you own a Pentium, is it possible, and efficient, to work around the bug with software? Or should you take advantage of Intel's offer to replace the chip?

The short answer is this: there is a very small chance of encountering arithmetic errors on the Pentium that are many orders of magnitude larger than they should be. If you want to have more confidence in the arithmetic operations done by the Pentium, you should replace the chip. If that's inconvenient, or while you are waiting for the replacement, you should use compilers and application software that incorporate the workaround described in this article. If you continue to use pre-Pentium software with an original-issue chip, your computations will probably not be affected, but you won't be able to tell for sure.

The story unfolds

The first signs of the Pentium FDIV bug were seen early last summer by Intel engineers in California and Oregon and by a mathematics professor in Virginia. (FDIV is the collective term for several floating-point divide

1070-9924/95/$4.00 © 1995 IEEE

instructions.) Intel became aware of the problem while developing the Pentium's not-yet-announced successor. Since extensive testing of the Pentium itself had not revealed any difficulties, and since no sign of the bug had been seen elsewhere, Intel was convinced that the problem was almost inconsequential. H.P. Sharangpani, an Intel chip designer, and M.L. Barton, an Intel computational scientist, carried out an extensive study during the summer of 1994. However, their report, "Statistical Analysis of Floating Point Flaw in the Pentium Processor (1994)," was not made public until the middle of December, by which time the bug had most definitely been seen elsewhere.

Nicely's prime

Thomas Nicely is a mathematics professor at Lynchburg College in Virginia. His research involves computational number theory, especially the distribution of prime numbers. Among all integers, the primes, p_n, occur infrequently. But they occur often enough to make the infinite sum of their reciprocals, $\Sigma\ 1/p_n$, diverge. Twin primes, t_n, are pairs of consecutive odd numbers, like 29 and 31, which are both prime. It is not known whether there are infinitely many twin primes, but even if there are, they occur so rarely that the sum of their reciprocals, $\Sigma\ 1/t_n$, has a finite value. This quantity is known as Brun's sum, but it has never been evaluated very accurately. Nicely set out to compute it. He has been working on the project for over a year, using half a dozen different computers and an extensive collection of software that evaluates several interrelated quantities. In March 1994, he added a Pentium to his collection of machines, and in June, he noted inconsistencies between values he was computing using hardware-based floating-point arithmetic and software-based multiple-precision arithmetic. It took him four months to confirm that the difficulty was not in his own programs, not in his math libraries, not in his compiler, not in his operating system, but in the hardware itself.

The pair of 12-digit integers, p and $p + 2$, where

$$p = 824633702441$$

are twin primes. To 16 significant figures, the reciprocal of p is

$$1/p = 1.212659629408667 \cdot 10^{-12}$$

But, the value of the reciprocal computed by the Pentium is

$$1/p \approx 1.212659624891158 \cdot 10^{-12}$$

The error is after the ninth significant digit. Errors of this size occur all the time in floating-point computation. But, before the Pentium, they could always be traced to the use of the 32-bit, single-precision floating-point format, not the 64-bit, double-precision and 80-bit, extended-precision formats that Nicely was using. (See the sidebar on IEEE floating-point arithmetic.) Nicely reported the problem to Intel Customer Support, but did not get an immediate response.

IEEE Floating-Point Arithmetic

The ANSI/IEEE Standard 754 for Binary Floating-Point Arithmetic, approved 10 years ago this month on March 21, 1985, is the basis for the floating-point architecture of all contemporary computers. Any nonzero number is written in the form

$$\pm (1 + f) \cdot 2^e$$

with $0 <= f < 1$ and integer e. The quantity f is the *fraction*, or, less officially, the *mantissa*. The quantity e is the *exponent*. Both f and e must have finite expansions in base 2. The finite nature of f means that numbers have a limited *accuracy* and so arithmetic may involve *round-off error*. The finite nature of e means that numbers have a limited *range* and so arithmetic may involve *under-* and *overflow*.

Let p be the number of bits used to represent f. Then $2^p \cdot f$ is an integer. Between any two successive powers of 2, say 2^e and 2^{e+1}, there are 2^p floating-point numbers, equally spaced a distance 2^{e-p} apart. The quantity 2^{-p}, which is the distance from 1 to the next larger floating-point number, is called the *relative accuracy* or one *unit-in-the-last-place* and is often denoted by ε.

The standard allows some flexibility in the number of possible formats and the details of the representation. The instance of the standard implemented on the Pentium and previous Intel microprocessors provides three formats:

	Single precision	Double precision	Extended precision
Word size in bits	32	64	80
Bits for f	23	52	63
Bits for e	8	11	15
Relative accuracy	$2^{-23} \approx 1.2 \cdot 10^{-7}$	$2^{-52} \approx 2.2 \cdot 10^{-16}$	$2^{-63} \approx 1.1 \cdot 10^{-19}$
Approximate range	$2^{\pm 127} \approx 10^{\pm 38}$	$2^{\pm 1023} \approx 10^{\pm 308}$	$2^{\pm 16383} \approx 10^{\pm 4964}$

For example, to express the decimal fraction 1/10 in base 2 requires an infinite series,

$$1/10 = (1 + 2^{-1} + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + \ldots) \cdot 2^{-4}$$

Floating-point approximations to 1/10 are obtained from 23, 52, or 63 terms of this series. Binary numbers are often printed by grouping four bits at a time into a base 16 or *hexadecimal* representation. The resulting approximations to 1/10 are

Single precision: `1.99999a · 2^-4`
Double precision: `1.999999999999a · 2^-4`
Extended precision: `1.999999999999999a · 2^-4`

On October 30, Nicely sent an e-mail memo to several other people he knew were doing technical computations on the Pentium. It was forwarded to Richard Smith, president of Phar Lap Software, who posted it to the Canopus forum on CompuServe, a commercial network devoted largely to PC users. Alex Wolfe, a reporter for the *EE Times*, a computer-industry weekly newspaper, spotted the CompuServe posting and sent it on to one of us—Terje Mathisen—the first of the authors of this article to get involved.

Wolfe sent the memo to Mathisen because Mathisen had, the day before, coincidentally posted a note to the Internet news group comp.sys.intel about the accuracy of the Pentium transcendental math functions. The Pentium uses different algorithms than earlier Intel chips. The results for functions like $\sin(x)$ have improved accuracy and may differ in the least significant bit of the double-precision representation. As a result, the Pentium actually fails a test program designed for the earlier chips. Mathisen's posting explained all this and pointed out that it was unimportant. But the error Nicely had uncovered in $1/x$ was far larger and completely unexpected.

On November 3, Mathisen posted an article with the title "Glaring FDIV bug in Pentium!" to comp.sys.intel. He used Nicely's prime and included a small test program. On November 7, the *EE Times* ran a front-page story by Wolfe with the headline "Intel Fixes a Pentium FPU Glitch." These stories set off a frenzy of Internet activity. At the height of the frenzy a month later, over 2,000 messages a week were being posted to comp.sys.intel, and articles about the Pentium or FDIV were being posted to over 300 news groups.

Kaiser's list

After Mathisen's posting, several people began experiments to assess the extent of the error. In a few days, Andreas Kaiser, a computer consultant in Stuttgart, Germany, generated 25 billion random integers and checked the accuracy of the computed reciprocals. He posted a list of 23 integers whose reciprocals are computed inaccurately by the Pentium.

The exponent portion of the floating-point representation is not involved in the error condition, so scaling by powers of two does not have any significant effect. Kaiser scaled his values to be integers to avoid the difficulties associated with binary conversion of decimal fractions. Figure 1 shows a portion of this list, plus the integers' hexadecimal floating-point representations. The first number, 3221224323, is the smallest integer whose reciprocal fails. None of these numbers can be represented exactly in single-precision floating-point format, so at this point it still appeared that the difficulty was limited to double precision.

The first thing to notice about Kaiser's list is the string of `f`'s in each hexadecimal representation. In fact, all but two of the numbers on the list start with `1.7ffff`. Each `f` corresponds to four ones in binary. So, in almost all of Kaiser's numbers, the first 20 bits after the leading bit have to be a single zero, followed by at least 19 ones. The two exceptions to this pattern on Kaiser's list have at least 12 ones near the beginning of the fraction. After these leading ones, the bit pattern is not so obvious, but it's still important.

```
3221224323 = 1.7ffff70600000 · 2^31
12884897291 = 1.7ffff70580000 · 2^33
206158356633 = 1.7ffff704c8000 · 2^37
824633702441 = 1.7fffff7052000 · 2^39
1443107810341 = 1.4fffedac25000 · 2^40
6597069619549 = 1.7fffff7057400 · 2^42
9895574626641 = 1.1fffc6bc2a200 · 2^43
13194134824767 = 1.7ffff704e7e00 · 2^43
26388269649885 = 1.7ffff704fdd00 · 2^44
52776539295213 = 1.7ffff7046f680 · 2^45
```

Figure 1. Testing 25 billion random integers, Andreas Kaiser found 23 whose reciprocals are computed inaccurately by the Pentium. Here is part of that list, with floating-point representations.

Coe's ratio

One of us, Tim Coe, was reading comp.sys.intel because he was thinking of buying his own Pentium-based computer. Coe, an electrical engineer, has designed floating-point chips. He was particularly intrigued by Kaiser's list of erroneous reciprocals, which provided a rare glimpse at the inner workings of another designer's creation. Coe was able to use the list to devise a model of the behavior of the Pentium division algorithm that explained its errors. Earlier Intel floating-point chips had used a fairly simple, binary, repeated-subtraction algorithm for floating-point division. But the Pentium has Intel's first implementation of a radix 4 SRT algorithm. SRT refers to the authors of the original algorithm, D. Sweeney, J.E. Robertson, and T.D. Tocher, whose work was published in 1958. The use of 4 for the radix, or number base, produces two bits in the quotient on each machine cycle, effectively doubling the performance over earlier chips running at the same clock rate.

Kaiser's and other early experiments had followed Nicely's example and concentrated on

computing reciprocals, $1/x$. But the FDIV commands, and Coe's model, accept a pair of floating-point arguments, x and y, and compute their ratio, y/x. Coe realized that the situations leading to the largest errors involved both x and y and bit patterns that "conspire" to excite the bug at an early stage in the division. He illustrated this with the example 4195835/3145727 in a posting to comp.sys.intel on November 14, 1994. The true value is

1.33382044...

But the value computed by the Pentium is

1.333*73906*...

The Pentium value is accurate to only 14 bits, or less than five significant decimal digits. The relative error is $6.1 \cdot 10^{-5}$. This is worse than single-precision round-off error and over ten orders of magnitude worse than double-precision round-off error.

On November 15 one of us, Cleve Moler, posted a summary of what was known at the time to comp.sys.intel, as well as to the Internet news groups devoted to numerical analysis and to Matlab, the mathematical software package put out by the MathWorks. He used Nicely's prime and Coe's ratio as examples. By this time, the Net had become hyperactive and the posting was redistributed widely. A week later, reporters for major newspapers and news services had Xeroxed copies of faxed copies of printouts of Moler's posting. At the same time, Moler began a series of postings about possible software workarounds for the bug and announced that the MathWorks would release a Pentium-aware version of Matlab that incorporated the workaround (see the sidebar on the Internet's role).

The story began to get popular press coverage when Net activists called their local newspapers and TV stations. On November 22, the Cable News Network was the first of the mass media to carry the story. Their report included brief interviews with Moler and with Stephen Smith, Intel's Pentium engineering manager. On Thanksgiving day, November 24, stories appeared in a number of major newspapers, including the *New York Times*, *Boston Globe*, and *San Jose Mercury News*. The *Times* story included a sidebar entitled "Close, But Not Close Enough," which used Coe's ratio to illustrate the problem. The story on page one of the *Globe* had the headline "Sorry, Wrong Number" and demonstrated the error by evaluating Coe's ratio in a spreadsheet.

The Role of the Internet

Bits, bugs, chips, and hexadecimals make up one side of this story. But looked at from a different perspective, the Pentium affair is also a story about the Internet. The initial publicity, the intense reaction, the source of media coverage, and the eventual corporate responses have all been on the Internet.

The authors of this article have never met face-to-face. Their collaboration is all by Internet, e-mail, and telephone. And they have offered the workaround to compiler vendors, commercial software developers, and ordinary folks via the Net. Certainly the facts would not have been discovered or disseminated as they were without this relatively new communication tool, which has now become a new medium for scientific investigation. —*Ed.*

By late November, Intel was offering to replace the chip for customers who could convince interviewers that they were heavy users of floating-point arithmetic. The Intel white paper by Sharangpani and Barton began to receive limited distribution outside the company. The first section of the paper outlines the SRT division algorithm and explains the defects in Pentium implementation. The description confirms the essential features of Coe's model.

The division algorithm

To get an idea of the source of the FDIV bug, consider the computation of Coe's ratio using long division and decimal arithmetic. After obtaining the first four significant digits of the quotient, the worksheet might look something like Figure 2 on the next page. What would you choose for the next digit in the quotient? The correct choice, when multiplied by the divisor, must produce a value close to, but less than, the remainder. When we do this by hand, we do not use precise algorithms for determining the digits; we use a combination of trial and error, experience, pattern matching, and luck. In this example, you can divide 25, the first two digits of the last remainder shown, by 3, the leading digit of the divisor, and correctly guess that the next digit in the quotient must be an 8.

The Pentium would do the same computation using radix 4, rather than decimal, arithmetic. In radix 4 notation, the normalized numerator and denominator are

$4195835 / 2^{22} = 1.00000113323$
$3145727 / 2^{21} = 1.13333333332$

The SRT algorithm constructs the quotient from the digits −2, −1, 0, +1, and +2. Using

```
                    1.333?
            ------------------
3145727 ) 4195835
          3145727
          -------
           10501080
            9437181
           --------
            10638990
             9437181
            --------
             12018090
              9437181
             --------
              25809090
              ????????
```

Figure 2. Computing Coe's ratio using long division and decimal arithmetic. The first four significant digits of the quotient have been found. What comes next?

```
          q+ =  1.0000002?
          q- = -0.1111110?
          ------------------
1.13333333332 ) 1.00000113323
                1.13333333332
                -------------
                 -133332200030
                 -113333333332
                 -------------
                  -133322000320
                  -113333333332
                  -------------
                   -133220003220
                   -113333333332
                   -------------
                    -132200032220
                    -113333333332
                    -------------
                     -122000322220
                     -113333333332
                     -------------
                      -020003222220
                      -113333333332
                      -------------
                       333301111120
                       233333333330
                       ------------
                        333011111300
                        ????????????
```

Figure 3. Computing Coe's ratio using the radix 4 SRT algorithm. The divisor, dividend, and partial remainders are represented in base 4, with digits 0 through 3. But the quotient has a nonunique representation with positive digits in q+ and negative digits in q–. The first eight steps are shown. What is the next quotient digit? This is where the Pentium goes wrong.

five digits with radix 4 leads to a nonunique representation, but allows the digits to be chosen quickly. Figure 3 shows the first eight steps of the algorithm. Even though the normalized numerator is less than the denominator, the algorithm chooses +1 as the first digit. This leads to a negative remainder and a choice of –1 as the second digit. This is repeated five more times. Then the remainder becomes positive and +2 is chosen as the eighth digit in the quotient. It is now time to choose the ninth digit. The correct choice would be another +2, but this is where the Pentium makes the error.

The bug: A faulty lookup table

The quotient digits are obtained by table lookup in a two-dimensional array. One index into the table, obtained from the first five bits in the divisor, is fixed throughout the division. For this example, the quotient index, in binary, is 1.0111. The first bit, which is always 1, is skipped. The other four bits indicate that the seventh column in the table should be used. The other index changes at each step and is generated by the first seven bits of the *carry-save* representation of the remainder. This is a nonunique representation of the remainder as the sum of two quantities, the *sum word* and the *carry word*, which reduces the time required by the carry operations generated during addition. In the SRT algorithm, the absolute value of the remainder is bounded by 8/3 of the divisor. For this example, the maximum remainder would be represented in radix 4 by a long string of 3's. This maximum was nearly reached on the eighth step in the example, but it turns out that the first seven bits of the carry-save representation are only 0011110. At the ninth step, the remainder is actually slightly smaller, but the first seven bits of its carry-save representation are 0011111, which is the maximum value. *The table entry associated with this extreme case is missing.* The last active entry in five columns of the SRT quotient digit table was omitted when the table was implemented in the Programmable Lookup Array on the chip. In effect, a zero is used instead of the +2. Nothing is subtracted from the remainder. The subsequent multiplication by 4 overflows the register and, although the algorithm proceeds to generate several more digits, it cannot recover.

The quotient digit table has 16 columns indexed by the first four bits after the leading bit in the divisor. The bound on the size of the remainder implies that the active portions of these columns are not all the same length. The last

active entry in columns 1, 4, 7, 10, and 13 is missing. In hexadecimal notation, the indices of these defective columns are 1, 4, 7, a, and d. We call a floating-point number an *at-risk* divisor if it contains a sequence of bits that might lead to referencing one of the missing entries in the SRT quotient digit table. A precise characterization of at-risk divisors is hard, but there is a necessary condition that is easy to test. The first eleven bits, including the leading bit, must be one of these five sequences:

```
1.0001111111
1.0100111111
1.0111111111
1.1010111111
1.1101111111
```

In other words, the first eleven bits must be a one, followed by one of the five hexadecimal characters associated with the defective columns, followed by six more ones. Having an at-risk divisor certainly does not guarantee that there will be an FDIV error; the actual occurrence requires conspiring bit patterns in the numerator and denominator. But we can say that if the denominator does *not* contain one of these five bit patterns, an FDIV error will *not* occur.

A picture of the floating-point numbers between any two powers of two looks something like this:

```
[___( )_____( )_____( )_____( )_____( )___]
    1       4       7       a       d
```

There are five small subintervals, corresponding to the five defective columns in the quotient digit table. The length of each subinterval is 1/1024 of the overall length. Any division involving a divisor outside these subintervals is OK. And, only a small fraction of divisions involving divisors inside the subintervals produce erroneous results. For example, the hexadecimal representation of the denominator in Coe's ratio, 3145727, is

$$1.7\mathtt{ffff}8000000 \cdot 2^{21}$$

The digit 7 places this number in the third defective column. It is followed by 17 ones, far more than the six required to characterize it as an at-risk divisor.

Distribution of errors

It is helpful to visualize the errors as being laid out on the real plane, with a mark at each point (x, y) where the Pentium miscalculates y/x. A small portion of such a picture is shown in Figure 4, generated by Larry Hoyle of the University of Kansas. Since the IEEE floating-point standard employs signed-magnitude mantissas, the four quadrants of the plane behave identically with respect to division, allowing us to confine our attention to the positive (upper right) quadrant.

The condition characterizing at-risk divisors is complicated enough, but the condition characterizing a conspiring dividend which actually produces an error is hopelessly complicated. This is because the SRT algorithm holds the divisor fixed, whereas the dividend, in effect, changes each step in a nearly chaotic way. Basically, a dividend is at risk when at some point in the process it yields a partial remainder whose leading seven bits causes it to access the missing table entry for the column accessed by the divisor. This characterization cannot be used to decide whether a given dividend is at risk without a detailed model of the Pentium division hardware (or a working Pentium), which is beyond the scope of this article. However, it is possible to exhaustively test all possible single-precision quotients and to carefully sample double-precision and extended-precision quotients.

The bit patterns characterizing the at-risk divisors are 11 bits wide. So, for example, in the broad band [1024, 2048), the five at-risk bands have unit width. Furthermore, almost all of the errors involve divisors with eight or more ones

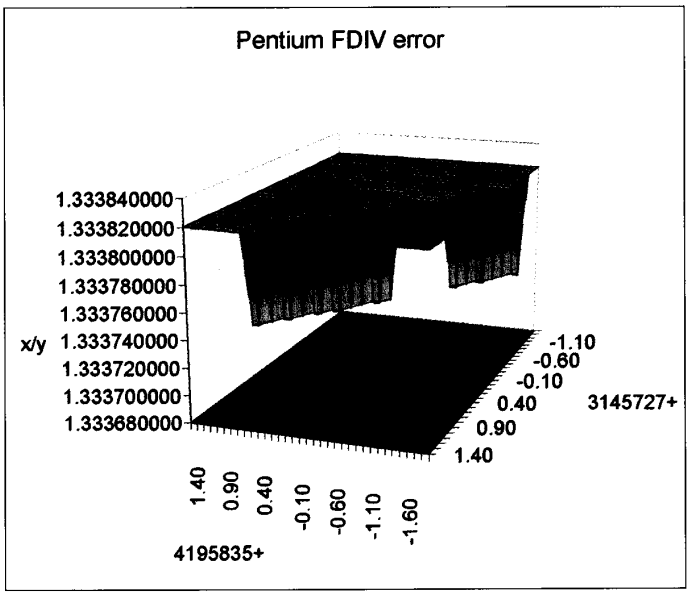


Figure 4. Larry Hoyle illustrated the range of numbers miscalculated by the Pentium chip using an Excel spreadsheet. He divided 4195835 by 3145727 and then varied the numbers slightly. The bug is shown in the perforations through the upper surface, which corresponds to correct results.

after the characteristic 1, 4, 7, a, or d, so the errors turn out to be concentrated near the right edge of each of these bands. One experiment involved all the single-precision denominators in these narrower bands, together with all the single-precision numerators between two successive powers of two. The results are summarized in the first row of Table 1.

The first row lists the errors actually observed for single-precision operands. The next two rows are extrapolated from the first under the plausible hypothesis that errors occur at the same rates at all operand precisions. The results are tabulated by the size of the error in the quotient; any error large enough to affect single precision is classified as a single-precision error, and so on.

A total of 9,915 errors were observed for single-precision operands. We expect this figure understates the true amount by nearly 600 owing to the difficulty of observing errors in the least-significant bit of extended precision. Of the 9,915 errors, 7,863 were larger than the error attributable to working at double precision, and hence should count as errors when performing double-precision arithmetic. And 1,738 of these errors were larger than the error attributable to using single precision. (Our experiments ignore the floating-point exponents and so do not take into account the possibility of underflow or overflow. Our table should therefore be interpreted as applying to the "normal operating range" in which these extremes are not encountered.)

The numbers on the diagonal in Table 1 are good estimates of the number of errors that would be counted in an experiment that involved all possible numerators and denominators in the interval between two successive powers of two. It would take too long to run such an experiment, but we have accomplished practically the same thing by looking only at single precision, and looking only at at-risk denominators. The total number of operands in such an experiment would be $(2^{23})^2$, $(2^{52})^2$, and $(2^{63})^2$, which are roughly

$7.037 \cdot 10^{13}$ for single precision,
$2.028 \cdot 10^{31}$ for double precision, and
$8.507 \cdot 10^{37}$ for extended precision.

The *probability* of an FDIV error can now be obtained by dividing the counts on the diagonal in Table 1 by the total number of possible operands. The *error rates* are the reciprocals of the probabilities:

	Probability	Rate
Single:	$2.470 \cdot 10^{-11}$	$40 \cdot 10^9$
Double:	$1.117 \cdot 10^{-10}$	$9 \cdot 10^9$
Extended:	$1.409 \cdot 10^{-10}$	$7 \cdot 10^9$

By coincidence the rates are all within a percent of being an integer multiple of billions. One in about 40 billion operand pairs delivers a single-precision error, one in about 9 billion a double-precision error, and one in about 7 billion an extended-precision error.

The 9,915 errors turn out to be quite uniformly distributed. Table 2 gives the number of single-precision operand pairs for a given cycle at which an error occurs, listed by row, and for a given divisor column, listed by column. The sum of the $(34 - 8) \cdot 5 = 130$ entries is 9,915.

From considerations of errors made by the Pentium in failing to correctly round certain otherwise correct quotients, we have deduced that the Pentium's implementation of the SRT algorithm runs for exactly 34 cycles. In searching for errors we set the threshold of observed error too high to detect errors in the 34th cycle, or in the 33rd cycle in the case of the first column of divisors, and there are probably a few other omissions from cycle 33. Based on the lower half of the table we would expect cycles 33 and 34 to contribute about 410 errors each, which would bring the total number of single-precision errors to approximately 10,490.

Table 1. The number of dividend/divisor pairs that cause Pentium errors, by precision of operands and size of error (what level of precision the resulting quotients failed to meet).

Operand precision	Quotient error level: Single	Double	Extended
Single	1,738	7,863	9,915
Double	$5.009 \cdot 10^{20}$	$2.266 \cdot 10^{21}$	$2.858 \cdot 10^{21}$
Extended	$2.101 \cdot 10^{27}$	$9.502 \cdot 10^{27}$	$1.199 \cdot 10^{28}$

Distribution of residuals

The Pentium division errors have a particularly simple form when viewed in terms of the residual, $y - (y/x) \cdot x$. The residual is always a power of 2 (that is, ..., 0.25, 0.5, 1, 2, 4, ...) except when the divisor x occurs in the first column (begins with 1.0001111111), in which case it is three times a power of two (..., 0.75, 1.5, 3, 6, 12, ...). This empirically observed behavior can be deduced from the values of the divisor where the error can occur and the fact that the missing table entries are just below the boundary $y = (8/3)x$.

Let's consider the case involving the first defective column with, for example, $x = 18$, $y = (8/3) \cdot 18 = 48$. The SRT algorithm should subtract $2x$ from y yielding 12, and multiply by 4 yielding 48 again. Instead it skips the subtraction, and the multiplication by 4 yields 192, at a scale that puts this quantity in an 8-bit register and makes 128 the sign bit. Hence the algorithm sees −64, an impossible value for which the table gives the quotient the digit 0. At the next cycle a further multiplication by 4 drives the 192 out of the register altogether, and it disappears unnoticed. Equivalently we may say that $192 \cdot 4$, which equals $3 \cdot 256$, is mistaken for zero two cycles after first encountering a missing entry, making the error 768 at that stage, equivalent to an error of 192 at the previous cycle. The total sequence of quotient digits determined by the algorithm is identical to what it would have been had 192 (scaled by the appropriate power of two) been subtracted from the dividend in the beginning.

When $x = 21$, $y = 8/3 \cdot 21 = 56$. Multiplying y by 4 yields 224, which the algorithm mistakes for −32. This is in the permitted range for this divisor, namely −56 to 56, and the algorithm proceeds normally. The loss therefore is $224 - (-32)$, or 256, one cycle after encountering the missing entry.

When $x = 24$, $y = (8/3) \cdot 24 = 64$. Multiplication by 4 yields 256, which the algorithm mistakes for 0, well within the permitted range, and hence we have lost 256 in the same way as for $x = 21$. (This is the error we saw at the ninth step of the worked example above.) The corresponding calculations for $x = 27$ and 30 yield the same conclusion: the algorithm recovers in one cycle and effectively loses 256 from the partial remainder.

For the general case, when y is a small distance below the $(8/3)x$ boundary, the missing table entries permit the sampled partial remainder to be at most two less than the boundary case (at a scale that makes 128 the sign bit). (This is true whether the partial remainder is represented in binary or, as on the Pentium, in redundant carry-save format.) This is not enough of a departure from the boundary case to change the fact that the algorithm recovers in two cycles for x in the 18 column and in one cycle otherwise. In all cases the error is caused by mistaking one multiple of 256 for another, in the first case yielding an error of exactly $3 \cdot 256$, in the remaining four exactly 256 (or $4 \cdot 256$ when measured in the same cycle that the $3 \cdot 256$ is measured in), even when the dividend is slightly less than 8/3 the divisor.

Table 2. Distribution of single-precision errors, by SRT cycle number and defective column number. Early cycle numbers produce larger errors.

	Errors, by divisor column					
Division cycle	**18**	**21**	**24**	**27**	**30**	**Total**
9	96	35	16	12	16	175
10	128	43	95	28	26	320
11	108	52	96	56	102	414
12	121	46	88	45	94	394
13	139	51	99	51	95	435
14	140	54	91	45	95	425
15	128	54	109	52	97	440
16	139	52	99	50	86	426
17	136	54	104	47	98	439
18	138	54	102	47	91	432
19	134	57	104	37	85	417
20	140	46	103	44	83	416
21	127	50	101	38	89	405
22	132	54	100	45	68	399
23	140	49	95	43	92	419
24	138	53	101	49	89	430
25	134	50	99	46	91	420
26	131	53	95	42	82	403
27	132	48	94	41	94	409
28	134	51	95	40	86	406
29	137	50	92	46	93	418
30	134	55	100	39	85	413
31	128	54	102	41	80	405
32	135	53	96	40	83	407
33	0	48	81	46	73	248
34	0	0	0	0	0	0
Total	**3,149**	**1,266**	**2,357**	**1,070**	**2,073**	**9,915**

Bruised integers

Computations that divide only integers having together at most nine digits are free of errors, as far as we have been able to tell. The shortest integer examples of the error we have seen are 7/402651871, 7/805303742, 57/50331637, 383/3145725, 383/6291450, 861/4718589, and 861/9437178.

At the other extreme, small *bruised* integers are at far greater risk. Choose two integers from 1 to 100 at random and *bruise* them by subtracting one millionth from each. For example, the quotient 5/15 becomes

$$\frac{4.999999}{14.999999}$$

The correct quotient starts with six 3's, 0.33333329, but the Pentium's value has only four 3's, 0.3333*2922*.

If one of these bruised integers is divided by the other, the chances of encountering a cycle

10 error (the second largest possible) is .08 percent. For a cycle 11 error it is .15 percent, and for cycle 12, .17 percent. Compare these figures with those for random data. The chance of a cycle 10 error for operands with uniformly distributed mantissas is one in 218 billion, a tiny fraction of the one in 9 billion double-precision errors. The error rate for small bruised integers is more than eight orders of magnitude higher!

Even worse rates are possible under certain circumstances. Suppose that before performing the division one multiplies the first number by the second and the second by 3. For example if the bruised integers were 6.999999 and 1.999999, their product would be 13.999991 and the second times 3 would be 5.999997, and in this example 13.999991/5.999997 happens to produce a cycle 10 error, corresponding to a relative error of a little under one in a hundred thousand. With this procedure the error rate increases to .21 percent, 1.13 percent, and .68 percent for cycles 10 through 12 respectively. So more than one in a hundred divisions of operands produced in this way have a relative error larger than one in a million.

Slightly larger bruising can produce cycle 9 errors, the maximum possible. If instead of bruising by subtracting a millionth we subtract one over a hundred thousand, the procedure of the preceding paragraph has a .1 percent chance of producing a cycle 9 error, corresponding to 10 such errors out of 10,000 possible pairs of operands.

The FDIV bug's effect on other instructions

In addition to the floating-point division instructions, the Pentium's faulty division circuitry affects three other groups of instructions: remainder, tangent, and arctangent. The remainder function is defined by

$$\text{rem}(y, x) = y - nx$$

where n is the integer part of y/x. If the quotient y/x is inaccurate and so large that the error occurs to the left of the decimal point, then n and hence the final result will be incorrect. An example can be obtained by scaling the numerator of Coe's ratio:

$$\begin{aligned} r &= \text{rem}(2^{15} \cdot 4195835, 3145727) \\ &= \text{rem}(137489121280, 3145727) \end{aligned}$$

The correct value is $r = 1977018$, but the Pentium-computed value is $r = 11414199$.

The Pentium has instructions for evaluating a number of transcendental mathematical functions. Because division is the slowest arithmetic operation, many of the algorithms were designed to avoid divisions altogether. So the computation of functions like $\exp(x)$, $\sin(x)$, and $\cos(x)$ does not use the defective division circuitry. The computation of the logarithm, $\ln(x)$, does involve a division, but only of numbers that avoid the at-risk denominators. Of all the transcendental functions, only tangent and arctangent are affected by the faulty division.

Tangent. The tangent function, $\tan(x)$, is evaluated by first safely computing $\sin(x)$ and $\cos(x)$. But then, the final division, $\tan(x) = \sin(x)/\cos(x)$, can fail. The failure is extremely rare because x must be a value for which $\cos(x)$ is an at-risk denominator and $\sin(x)$ is a conspiring numerator. But such values do exist. One example is

$$x = 0.8549592142878324$$

The correct value of $\tan(x)$ is

$$\tan(x) = 1.149782816566037$$

The Pentium-computed value is

$$\tan(x) = 1.149782816388642$$

The relative error is about $1.5 \cdot 10^{-10}$. In hexadecimal,

$$\cos(x) = 1.4\text{fffe}80161\text{df}7 \cdot 2^{-1}$$

shows an at-risk pattern.

Arctangent. The Pentium's FPATAN instruction evaluates the four-quadrant arctangent function, which adds multiples of π to $\arctan(y/x)$ to reflect the Cartesian quadrant of the point (x, y). We don't know all the details of the Intel proprietary algorithm, but it is probably based on an identity like

$$\arctan(y/x) = \arctan t + \arctan\left(\frac{y - tx}{x + ty}\right)$$

The first term on the right could be obtained from a table, stored on the chip, of a few dozen values of $\arctan(t)$. The second term is a small correction, which can be computed quickly. For each x and y the value of t closest to y/x is chosen, but the computed value of y/x is not actually used. The only division that affects the final accuracy is the computation of the argument of

the correction term. But an inaccurate division is possible here. For example, with y = 0.9333324451392881 and x = 1 the correct value is

$$\arctan(y/x) = 0.7509285877097011$$

But the Pentium computes

$$\arctan(y/x) = 0.7509285896965135$$

This can be explained by taking t to be 15/16 = 0.9375. The crucial division would be

$$\frac{(y-t)}{(1+ty)} = \frac{-0.0041675548607119}{1.874999167318082}$$

In hexadecimal, the denominator, `1.dffff207a88cb`, shows an at-risk pattern.

How will the errors affect technical computations?

The simplest model of the FDIV bug assumes that mantissas are uniformly distributed, in which case the above-mentioned error rates of one in 40, 9, and 7 billion divisions for respectively single, double, and extended precision hold. This is the only case considered by the Intel white paper describing the bug and assessing its impact. The most widely quoted conclusion reached in this paper was the following:

> In fact, extensive engineering tests demonstrated that an average spreadsheet user could encounter this subtle flaw of reduced precision once in every 27,000 years of use.

The 27,000 year figure follows from the nine billion double-precision error rate and a claim that a typical spreadsheet user does roughly 1,000 divisions per day, or about 1/3 million divisions per year.

If the erroneous operand pairs had been distributed at random, this model would be a reliable indicator for almost all applications, even those with far from random data. But as we have seen, the at-risk divisors cluster just below integers that are multiples of three, scaled by powers of two. This distinctly nonrandom behavior reduces the risk for certain applications, and increases it for certain others.

On December 12, IBM Corporation announced that they were suspending sales of any Pentium-based models of their personal computers. The IBM press release included the following paragraph:

> Intel has said that in purely random situations the likelihood of a customer encountering an error is only once in 27,000 years and that off-the-shelf software is not affected. However, IBM tests indicate that common spreadsheet programs, recalculating for 15 minutes a day, could produce Pentium-related errors as often as once every 24 days. For a customer with 500 Pentium-based PCs, this could result in as many as 20 mistakes a day.

The IBM study substantiating this claim is available from their World Wide Web site. Their analysis questions the assumption of uniformly distributed operand pairs and points out that certain types of operations on numbers expressed in decimal notation with two digits after the decimal point are likely to produce what we have called bruised integers. They claim it is reasonable to assume the probability of encountering an error-producing denominator and numerator is 1 in 1,000 and 1 in 100,000, respectively, and hence that the FDIV error rate is "typically" 1 in 100 million. A separate argument, based on

(1) the Pentium's speed,
(2) a claim that 1 out of every 16,000 operations is probably a division, and
(3) an assumption that a typical spreadsheet user does 15 minutes of intensive computation per day

leads to an estimate of 4.2 million divisions per day. This is combined with their 1 in 100 million error rate to obtain the 24 days figure.

Who is right, Intel or IBM? We agree with William Kahan, a professor at Berkeley and primary architect of the IEEE floating-point standard. The original *New York Times* story on the FDIV bug quoted Kahan:

> These kind of statistics have to cause some wonderment. They are based on assertions about the probability of events whose probability we don't know.

Kahan is right. We don't know how the floating-point numbers involved in serious technical computations are distributed, we don't know what fraction of them trigger the FDIV bug, we don't know the distribution of the magnitude of the errors that would result and, as both the Intel and IBM documents make clear, we don't have any idea how frequently divisions occur in practice.

Even if we did know all these distributions and probabilities, it is our opinion that these simple probabilistic models are missing the point. Numerical analytic round-off error analysis, the IEEE floating-point standard, and everyone's confidence in the results of technical computation are all based on a model of float-

ing-point arithmetic which simply fails to hold on the Pentium.

The software workaround

One of the first ideas for a software replacement of the division instruction was to use the chip to compute an approximate reciprocal, $1/x$. This would be followed by a couple steps of Newton's method to improve the accuracy and then a multiplication by y to obtain the final result. There are two things wrong with this idea. It would be unnecessarily slow, and it would not be as accurate as we would like. Even if the Newton iteration generated the correctly rounded value of $1/x$, the multiplication of y by $1/x$ would not have the same round-off properties as a correct division, y/x.

Another suggestion would have followed each division $z = y/x$ by a residual calculation $r = y - xz$. If the residual were too large, the division could be redone in software by Newton's method. Again, this would be slow and not fully accurate. The software workaround we eventually adopted doesn't compute the residual and doesn't do any divisions in software.

Three authors of this article (Moler, Coe, and Mathisen) have been working with a group of hardware and software engineers from Intel, and Peter Tang, who is from Argonne National Laboratory but is on sabbatical in Hong Kong. We have developed, implemented, and tested software (and proved the algorithms correct) to work around the FDIV bug and also around the related bugs in the Pentium's on-chip tangent, arctangent, and remainder instructions.

The general approach to our FDIV workaround was presented in one of three articles in the *New York Times* on December 14, 1994. We've reproduced the *Times*' sidebar here because it is a rare example of the accurate exposition of mathematical algorithms in the popular press. The key to our workaround is the fact that the chip does a perfectly good job with division almost all the time. All we need to do is avoid the operands that lead to errors. In our workaround, the denominator is quickly tested before each FDIV is done; if it is not an at-risk denominator, the FDIV can be done safely. If it is, the workaround multiplies both numerator and denominator by 15/16. This does not change the quotient, but it takes the denominator out of the unsafe region and insures that the subsequent FDIV will be accurate.

With this approach, it is not necessary to test the magnitude of the residual resulting from a division. We know a priori that all divisions will produce correct results. If desired, an additional test can compare the result of scaled and unscaled divisions and count the number of FDIV errors that would occur on an unaided Pentium. The Pentium-aware version of the Matlab software offers this test, but it can be turned off for maximum execution speed.

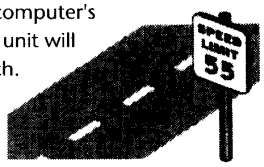
When an at-risk single- or double-precision denominator is encountered, the floating-point

Slow, Division Ahead

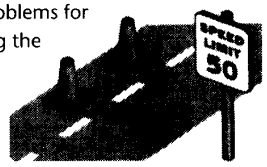
The widely-publicized flaw in Intel's Pentium microprocessor affects some operations involving the chip's floating point unit, which speeds up calculations that must be very precise. A piece of computer code called a software patch is being written to work around the problem. This patch will have to be added to popular software programs, like Microsoft Excel, and in some cases will slow down the performance of the computer.

The errors occur when the processor performs some division calculations. Computers use the binary system — 1's and 0's — instead of decimal numbers, so, for example, 22 is represented as 10110. In division problems where the result is not a whole number, the floating point unit consults a "look-up table" to find the closest binary equivalent. In most microprocessors the look-up tables are accurate to 16 digits, but some parts of the Pentium's look-up table are accurate to only 5 digits, far less precise than other chips.

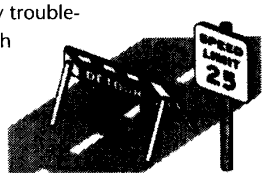
USUALLY, TOP SPEED Most of the computer's operations using the floating point unit will be unaffected by the software patch. The patch is only activated when the computer performs a calculation involving division.



RUBBERNECKING The patch slows the computer down a bit to inspect all division problems for a certain kind of number, containing the binary equivalent of the numbers 1, 4, 7, 10 or 13, followed by several binary 1's, in a certain location. If none are present, the computer resumes normal speed.



ALTERNATE ROUTE If the potentially troublesome numbers are present, the patch multiplies both parts of the division problem by 15/16. Adding this step will always force the computer to use an accurate part of the look-up table instead of the flawed part. But it makes the calculation take twice as long.



Courtesy Dylan McClain and Patrick J. Lyon of The New York Times

```
static char fdiv_bug;

long double fdiv(long double x, long double y)
{
  static char fdiv_risk[16] = {0,1,0,0,4,0,0,7,0,0,10,0,0,13,0,0};
  static float fdiv_scale = 0.9375;
  static float one_shl_63 = (65536.0*65536.0*65536.0*32768.0);

  if (fdiv_bug) {
    unsigned long t, *py = (unsigned long *) &y;
    do {
      t = py[1];
      if (t & 0x80000000) {
        t ^= 0x07e00000;           /* Invert six bits           */
        if (t & 0x07e00000) break; /* Does not have six 1 bits! */
        if (fdiv_risk[(t >> 27)-32] == 0) break; /* Not at-risk!*/
        x *= fdiv_scale;           /* 15/16 */
        y *= fdiv_scale;           /* 15/16 */
        break;
      }
      if ((t | py[0]) == 0) break; /* divide by zero!          */
      x *= one_shl_63;             /* Scale both operands by   */
      y *= one_shl_63              /* 1 << 63                  */
    } while (1);                   /* Restart denormal numbers! */
  }
  return x / y;
}
```

Figure 5. The complete FDIV workaround, written in C.

unit can be switched to extended precision. Both operands are then scaled by 15/16 exactly, using four of the 11 extra mantissa bits available internally. Then, when the resulting quotient is rounded back to the working precision, it will yield exactly the same result, down to the last bit, as would be produced by a chip without the FDIV bug.

We optimized the software implementation to give minimum performance impact on chips without the bug, while still keeping the overhead as low as possible on Pentiums with the bug. Figure 5 contains a C version of the complete workaround for FDIV. The algorithm works by first checking a global flag to see if the bug has been detected. If not, we jump directly to the FDIV instruction. Then, on a failing machine, we inspect the denominator bit pattern. This is done in extended precision, which reduces the need to handle denormal numbers. (If a program/compiler puts the floating-point unit in double precision, no denormal number can occur in extended precision, so those tests can be removed.) First we check if the leading mantissa bit is set. If not, we have either a denormal number or zero. If zero, we use the hardware to do the division by zero. If denormal we scale both operands by 2^{63} and restart the pattern testing. On a normal number, we first look for the six consecutive one bits. We do this not by masking and comparing, but with `Xor` and `Test`, which leave the rest of the bits unmodified for the remaining pattern checks. If all six bits are set, we then test the top four bits of the fraction using a software table lookup. The table has 16 entries, with a flag in the columns corresponding to the five missing table entries.

Unlike FDIV, there was no way to just inspect the input values to determine whether a given pair of operands for FPATAN might be at-risk, so we had to use a complete software replacement for this function. The resulting code uses extended precision for all intermediate operations, while the Pentium hardware instruction has access to internal paths with a couple of extra bits of precision. This means that while FPATAN, on a corrected Pentium, will always be less than 1 ULP (unit in the last place) away from the true result, our workaround can be up to 3 ULPs in error, in extended precision. However, when we round the result to double precision the replacement delivers results that are nearly indistinguishable from the true results, with a maximum error of 0.50146 ULP, versus FPATAN which gives about 0.50039 ULP. The software workaround gives about the same precision as the algorithm used by previous Intel numeric coprocessors.

To actually use the workaround in Figure 5, it would be necessary to modify application source code, replacing all divisions by calls to the new function. Most of the resulting increase in execution time would be caused by the overhead of the calling sequence. Moreover, some C compilers do not distinguish between the double-precision and extended-precision floating-point formats. With these compilers, the computed result might not be correctly rounded. To reduce the overhead and get fully accurate results, it is preferable to implement the workaround in assembly language and to use macro expansion to generate in-line code. Further discussion of this approach, and more information about the software workaround, is available with anonymous FTP from Intel's Internet site, ftp.intel.com, in file /pub/pentium/IAL/patch.txt.

Several compiler vendors have announced versions of their compilers which generate code using the workaround in place of the FDIV instruction, and which have math libraries incorporating the new algorithms for remainder, tangent, and arctangent.

When the workaround involves assembly code expanded in line, its cost in execution time is not high. The actual timing will depend on more factors than we want to consider here, but we expect it to involve a factor less than two. It should also be realized that few technical computations involve divisions in the inner loop, so the overall effect on execution time will almost always be very small.

On December 20, Intel announced a change of policy regarding Pentiums. It is now possible for anyone owning a chip with the FDIV bug to request a placement, either directly from Intel, from the manufacturer of the computer itself, or through specified local repair centers. The faulty chips must be returned to Intel.

How difficult is it to replace the chip? That varies from computer to computer. Many machines have Zero Insertion Force, or ZIF, sockets, so chip replacement is easy. Other machines have the chip permanently mounted on the main circuit board, and it is necessary to replace the entire board.

Should you replace your chip? In our opinion, yes. It is very unlikely your computations will actually encounter the bug. But, if they do, it is usually very difficult to detect, or to completely assess the consequences. If it is inconvenient for you to make the replacement, then you should at least try to use software which incorporates our workaround. ◆

For more information

The Pentium Papers, a collection of "primary source" material on the Pentium division bug, is available via the Internet from the MathWorks. On the World Wide Web, connect to http://www.mathworks.com. With anonymous FTP, access ftp.mathworks.com, and go to directory /pub/pentium. This archive is intended as a reasonably complete historical record of the events associated with the Pentium floating-point division bug. All of the documents are reproduced as they originally appeared on the Net or elsewhere.

Both Intel and IBM have Web pages with Pentium information: http://www.intel.com/product/pentium/fdiv.html, and http://www.ibm.com/Features/pentium.html. Intel's phone number for Pentium replacements is 1-800-628-8686.

Tim Coe is a principal engineer at Vitesse Semiconductor Corp. In the past, he has worked on the design of IEEE floating-point units, cache controllers, and multi-gigahertz multiplexors and demultiplexors. Currently, he is working on an implementation of the Scalable Coherent Interface (SCI). He received his BSEE and MSEE from MIT in 1986. He can be reached by e-mail to coe@vitsemi.com.

Terje W Mathisen is a systems architect in the Hydro Data division of Norsk Hydro, where he does low-level software development and optimization. He is interested in computational algorithms, and has written an unlimited-precision math library. Mathisen earned an MSc in electrical engineering in 1981 from the Norwegian Institute of Technology and is a member of the Norwegian Society of Charted Engineers. He can be reached by e-mail to Terje.Mathisen@hda.hydro.com.

Cleve Moler is chairman and chief scientist at The MathWorks, makers of Matlab (styled MATLAB by the company). Prior to this, he taught math and computer science at the Universities of Michigan and New Mexico and at Stanford, and worked at two computer hardware manufacturers, Intel Hypercube and Ardent Computer. His professional interests center on numerical analysis and mathematical software. In addition to being the author of the first version of Matlab, Moler coauthored the Linpack and Eispack scientific subroutine libraries and three textbooks on numerical methods. He holds a BS in math from Caltech, 1961, and a PhD in math from Stanford, 1965. Moler may be reached at the MathWorks, 24 Prime Park Way, Natick, MA 01760, e-mail moler@mathworks.com.

Vaughan Pratt is professor of computer science at Stanford University. From 1972 to 1982 he was on the electrical engineering and computer science faculty at MIT. He helped found Sun Microsystems in 1982, and designed Sun's logo and the Pixrect graphics system. He has worked in natural language, analysis of algorithms, program logic, concurrency modeling, computer graphics, and digital typography; his most recent research interest is in the foundations of mathematics and quantum mechanics. He received a BSc (honors) in 1967 and an MSc in 1970, both from Sydney University, and a PhD in 1972 from Stanford University. He may be reached by e-mail to pratt@cs.stanford.edu.