Anatomy of the Pentium Bug

Vaughan Pratt*
Dept. of Computer Science, Stanford University, Stanford, CA 94305-2140
pratt@cs.stanford.edu

July 26, 2005

Abstract

The Pentium computer chip's division algorithm relies on a table from which five entries were inadvertently omitted, with the result that 1738 single precision dividend-divisor pairs yield relative errors whose most significant bit is uniformly distributed from the 14th to the 23rd (least significant) bit. This corresponds to a rate of one error every 40 billion random single precision divisions. The same general pattern appears at double precision, with an error rate of one in every 9 billion divisions or 75 minutes of division time.

These rates assume randomly distributed *data*. The distribution of the faulty pairs themselves however is far from random, with the effect that if the data is so nonrandom as to be just the constant 1, then random *calculations* started from that constant produce a division error once every few minutes, and these errors will sometimes propagate many more steps. A much higher rate yet is obtained when dividing small (< 100) integers "bruised" by subtracting one millionth, where every 400 divisions will see a relative error of at least one in a million.

The software engineering implications of the bug include the observations that the method of exercising reachable components cannot detect reachable components mistakenly believed unreachable, and that hand-checked proofs build false confidence.

1 Background

The Intel Pentium[1] microprocessor makes occasional errors in floating point divisions due to five missing entries from a lookup table of quotient digits. Relative errors as large as 2^{-14} are possible. The errors depend only on the mantissas of the operands and not their exponents. In the space of all operand pairs, one in nine billion pairs generate an error in the quotient exceeding the usual double precision errors; of these, one in 40 billion generates an error exceeding single precision errors.

*This work was supported by ONR under grant number N00014-92-J-1974
[1]Pentium is a trademark of Intel Corporation

The error was first noticed within Intel and independently by Prof. Thomas Nicely at Lynchburg College, Virginia, in the course of an ongoing project to estimate the sum of the reciprocals of the twin primes, known to exist but not known to much accuracy. Nicely publicized his discovery at the end of October, and it quickly became a *cause célèbre* on the Internet.

In early December 1994 Intel put on their World-Wide Web home page their White Paper [SB94]. The paper described in broad outline the nature of the bug, and estimated that the average user would encounter the error once every 27,000 years.

Intel was initially reluctant to replace processors, and attempted to distinguish those with a genuine need. This proved unreliable and eventually Intel agreed to exchange processors with no requirement that any test be passed.

This paper gives a considerably more detailed account of the bug than can be inferred from the White Paper. We are indebted to Tim Coe, a floating point hardware designer at Vitesse Semiconductor for unearthing some of these details. Coe constructed a model of the bug that predicted most of the errors, and also found much larger errors than Nicely did, most notably 4195835/3145727 which the Pentium computes with a relative error of 2^{-14}. The bug renders visible many details of the Pentium's floating point division process that are invisible in the absence of the bug. In this respect the bug is like a linear accelerator with the errors being the analogue of scattered particles. We have duplicated Coe's model and modified it slightly to account for additional errors, exposing more architectural details.

A workaround for the bug has been developed by Terje Mathisen and Cleve Moler. The workaround is to detect those divisors at risk of an error and to scale both operands by a suitable constant to move the operands out of danger before performing the division. With this precaution the Pentium never performs an erroneous division, not even as an intermediate step in getting the correct result. These and other aspects of the bug have very recently been written up [CMMP95].

The natural software engineering question is, what did this bug teach us? For example, what could have been done differently that would have avoided this bug? The following points seem to us to be of general interest, not so much for their novelty as for their relevance to the Pentium bug.

1. An Achilles' heel for testing. One thinks of testing as being as good as verification *if* one could test all possible cases. As a weakened version of this, a comprehensive test should exercise every device and/or line of code in the system.

The Pentium bug reveals a serious limitation of this approach. There is of course no data that can exercise unreachable code or table entries. Thus if one believes that the five "missing" entries are unreachable, then no attempt will be made to produce a test for this case. Hence missing entries are likely to be overlooked by any *fabricated* set of test cases. Randomly generated test cases have a better chance of reaching a supposedly unreachable part of the system.

2. Manual verification has negative value. In contrast to computer proof checking, manual proof checking is a notoriously unreliable process. Neverthe-

less one may feel reassured after having proved manually that an algorithm works in every detail, and hence attach little incremental value to a machine-checked proof. This opens the way to $475,000,000 errors, Intel's estimate of the cost of the bug.

2 The SRT division algorithm

The SRT[2] division algorithm adapts the familiar process of long division to computer hardware, for an arbitrary radix $r \geq 2$. On binary computers (as opposed to say decimal), SRT happens to work particularly well at radix 4, which we now assume.

Division problems involving negative operands may be reduced to divisions involving only nonnegative operands, via the identities $(-y)/x = y/(-x) = -(y/x)$ and $(-y)/(-x) = y/x$. This lets us concentrate on nonnegative operands.

The basis for the radix 4 SRT algorithm is the following navigational strategy permitting a microbot, Robby, to determine his position to any desired accuracy when placed on the main diagonal of Figure 1.

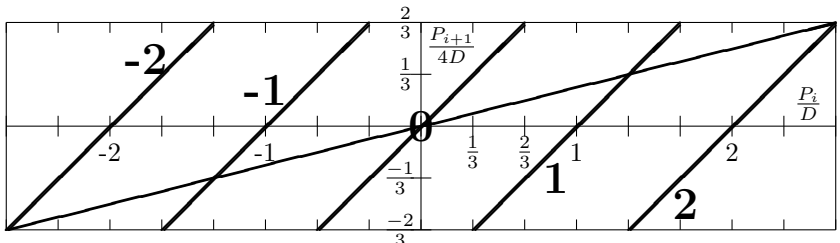

Figure 1. An easily navigated rectangle.

Robby walks around while writing down two nonnegative numbers R and L in radix 4 from left to right, such that at all times $R - L$ is his current estimate of the horizontal coordinate of his original position. Before writing the first digit, each number is understood to be zero, and the first digit is written in the units position, just before the point.

Robby begins by walking north or south so as to reach one of the five steep line segments (without leaving the rectangle), then writes one digit of each of R and L. One digit is zero, the other is the absolute value $|m|$ of the segment's label m, which if nonzero is appended to R or L according to whether the segment is on the right or left (positive or negative label) respectively (if $m = 0$ the choice is of course immaterial). Robby completes this cycle of his walk by returning to the main diagonal along a horizontal trajectory.

After c such cycles, Robby asserts that his original position was $R - L + e/4^c$ where e is the horizontal coordinate of his current position. The role of the $e/4^c$ term is as the error of his estimate of $R - L$ as his position.

Robby's assertion is proved by induction as follows. Initially $c = 0$, $R = L = 0$, and the current error is his initial position, verifying his assertion for the

[2]The SRT algorithm is named for its independent inventors Sweeney at IBM, Robinson at the University of Illinois [Rob58], and Tocher at Imperial College [Toc58].

basis case of the induction. Now assume that the assertion holds at cycle c, and suppose that label m is encountered on the next cycle. It suffices to show that the estimate increases by $m/4^c$ and the error decreases by that amount.

For the former, appending one of m or $-m$ at the c-th position of one of R or L respectively increases $R-L$ by $m/4^c$ in either case, whence $R-L$ increases by $m/4^c$.

For the latter, imagine Robby detours (still traveling vertically) to segment 0 (extended if necessary) on the way to m. Leaving from $(e, e/4)$, he must arrive at (e, e) on segment 0 due to its 45 degree slope. Segment 0 being at height m above segment m, he reaches the latter at the point $(e, e-m)$. The trip back to the main diagonal simply sets the horizontal coordinate to four time the vertical, so his position at the end of the cycle is $(4(e-m), e-m)$. This completes cycle $c+1$, making his error his new position $4(e-m)$ divided by 4^{c+1}, namely $(e-m)/4^c$. This is $m/4^c$ less than before, and with this we have completed the proof of the assertion.

Given a dividend P_0 and a divisor D, aligned so that P_0 lies in $(D/2, 2D)$, the Pentium solves the problem of estimating P_0/D by the above method. The division process begins by loading P_0 into a register P and initializing register R and L to zero. Before each cycle of the division P/D corresponds to the horizontal coordinate of the position on the diagonal. The vertical motion to segment m is accomplished by forming mD (with at most a shift and a negation) and subtracting it from P, which changes P/D to $P/D - m$. The horizontal motion is accomplished by shifting P left two bits to make the new position $4(P/D - m)$. The writing of R and L is accomplished with two registers; to append a digit, shift the register left two bits (the point may be understood as shifting too) and store the digit in the two low order bits. The Pentium carries out this process in exactly 34 cycles, enough for the 64 bits of an extended precision mantissa plus the guard, round, and sticky bits needed to perform correct IEEE rounding. Hence when done, $R - L$ as an estimate of P_0/D to within $\frac{8}{3}/4^{34} = 1/(3 * 2^{65})$ is the desired quotient.

The hard part is to determine m. This is accomplished by computing an approximation to P/D by sampling a few of the leading bits of P and D to yield P' and D', the *chopped* partial

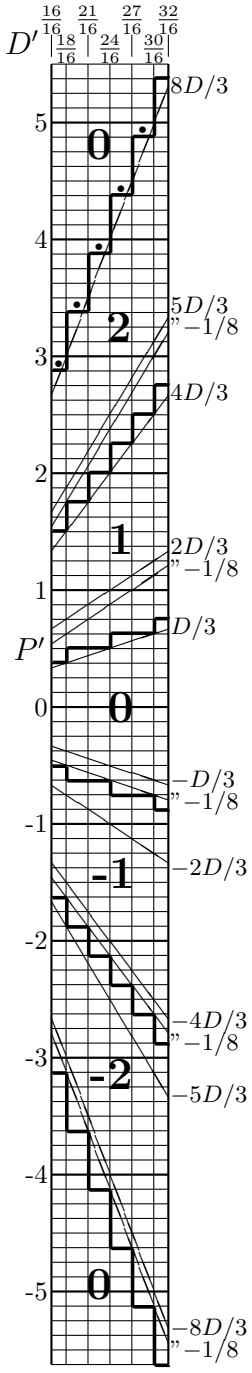

Figure 2.
The PD-plot

remainder and divisor respectively. We then use these to index a table of integer approximations to P'/D' called the *PD-plot* [Atk68]. According to the White Paper [SB94] the Pentium chops P down to $P' = \lfloor 8P \rfloor/8$, and chops D down to $D' = \lfloor 16D \rfloor/16$. These approximations correspond to sampling the first 7 significant bits of P, the first bit being the sign bit and the binary point following the first four bits, and to sampling the first 5 significant bits of D, this being the number of bits in an integer in $[16, 32)$.

Since D is in $[1, 2)$, it follows that P is in $[-16/3, 16/3]$ (consider D near 2), whence $\lfloor 8P \rfloor$ ranges from $\lfloor -128/3 \rfloor = -43$ to $\lfloor 128/3 \rfloor = 42$, a total of 86 distinct values. And for D in $[1, 2)$, $\lfloor 16D \rfloor$ ranges from 16 to 31, a total of 16 distinct values.

Coe has conjectured to us that the Pentium approximates D more coarsely than this by taking $D' = 3\lfloor 16D/3 \rfloor/16$, on the ground that it works and makes the table nearly one-third the size. This has the effect of coalescing the 16 samples in twos and threes to form six groups, corresponding to the six possible values of $\lfloor 16D/3 \rfloor$ when $D \in [1, 2)$. We assume Coe's conjecture for the sequel.

The Pentium's PD-plot is depicted in Figure 2 as an 89×6 array of cells (this may be one or two more rows than are physically present in the Pentium).

The P and D dimensions are both drawn at a scale of one unit per 15mm. Cells hold integers in $[-2, 2]$, and are grouped into homogeneous regions in each of which all cells contain the same value. (The cells would reasonably be coded with two bits, with the sign of the entries implicit and inferrable from that of P'.) The regions are delimited in the diagram by the bold jagged lines, called *thresholds*, and the common value in the cells of a region is given by the large boldface number in the center of that region.

The PD-plot can be viewed simultaneously as a discrete physical table indexed by P' and D' and as a continuum (to within extended precision) indexed by P and D. This permits the chopping process to be visualized as taking place in the diagram: chopping P moves it to the lower edge of the containing cell while chopping D moves it to the left edge.

The crucial property is that *the m obtained with P' and D' from the table is such that P lies in the interval* $[(m - 2/3)D, (m + 2/3)D]$. These five intervals determine 10 boundaries, which are depicted in the figure as the lines

$8D/3$ and so on (ignore for now the lower line of any pair of parallel lines 1/8 apart). The crucial fact proving this property is that *the thresholds lie in the intersection of their corresponding intervals.* The Pentium's PD-plot illustrates this nicely with five correct examples and one incorrect one, the top one, which is the bug.

For those locations corresponding to no possible P and D one arbitrarily sets m to whatever minimizes power consumption, apparently $m = 0$.

Complicating all this greatly is that the number P is represented in the Pentium in carry-save form, namely as a pair S, C of numbers satisfying $S + C = P$. This representation has the property that a quantity such as the divisor can be added or subtracted in constant time independent of the word length, significant for 68-bit words.

Initially S is set to P_0 and C to 0. To add D to P, form $S = S \oplus C \oplus D$ (exclusive or), $C = (S \wedge C) \vee (C \wedge D) \vee (D \wedge S)$ in parallel, that is, both right hand sides are evaluated before performing either assignment to the left hand side. Then shift C left one place. To subtract D, add the logical complement of D, that is, $-D - 1$, as above. Then compensate for the -1 by setting to 1 the low order bit of C, which had been cleared to 0 by the shift.

The underlying principle is that these bit-vector operations implement a full adder at every bit position. A full adder has three inputs, and its two outputs, called *sum* and *carry*, record as a two-bit binary numeral the number of 1's on its inputs. The sum bits form S and the carry bits C.

This complicates the computation of m. We now compute P' as $(\lfloor 8S \rfloor + \lfloor 8C \rfloor)/8$. Previously $P' \leq P < P' + 1/8$. The redundant carry-save representation weakens this to $P' \leq P < P' + 1/4$. A simple yet rigorously justifiable way of accommodating the additional 1/8 is to lower by 1/8 *every boundary serving to bound a threshold from above but not from below.* All but the uppermost threshold have such boundaries, leading to the five pairs of parallel lines in Figure 2. Observe that even after so narrowing the regions of uncertainty the thresholds still lie completely within those regions, the crucial correctness property for the PD-plot.

It should be pointed out that the technique of accumulating the positive and negative contributions to the quotient in separate registers R and L constitutes a redundant representation of the quotient serving the same purpose, namely to avoid having to wait for carries to propagate.

The thresholds of Figure 2 cannot be inferred directly from the Intel White Paper's explanation. However using the bug described below as a "linear accelerator" it becomes possible to test whether any other settings of these thresholds predicts the errors experienced by the Pentium. With the exception of the uppermost threshold, which is low by 1 in the Pentium, all other settings of these thresholds turn out to generate errors in the model that are incompatible with the Pentium's observed behavior. In making these measurements one can also

measure the length of the P register as having at least 68 bits counting the sign bit. It is not possible however to use it to test Coe's hypothesis that there are only six "chopped" divisors D'; the bug cannot distinguish between Coe's hypothesis and the 16 chopped divisors shown in the White Paper.

3 Nature of the bug

The bug is that the top threshold in Figure 2 is set one position too low. This sets to 0 five accessible entries in the table that should have been 2 (the sixth can remain zero because $D < 2$), dotted in Figure 2. The White Paper attributes the error to a script that incorrectly copied values; one is nevertheless tempted to wonder whether the rule for lowering thresholds was applied to the $8D/3$ boundary, which would be an incorrect application because that boundary is serving to bound a threshold from below.

The effect of taking the quotient digit to be 0 when it should be 2 can be understood by first considering P to be the boundary case $8D/3$ and D to be one of the boundary cases 18/16, 21/16, 24/16, 27/16, and 30/16. While these cases just miss the erroneous entries, there are cases that hit these entries that approach these boundary cases arbitrarily closely. The statistical properties of carry-save arithmetic are such that to reach a missing entry D must lie in $[T - 2^{-10}, T)$ where $16T$ is an integer multiple of 3. Since $D \in [1, 2)$ there are only five possible such T's, namely $16T = 18$, 21, 24, 27, or 30.

For $D = 18/16$, $P = 8/3 \cdot 18/16 = 3$. This quantity should have been reduced to 3/4 by subtraction of $2D = 9/4$. Instead nothing is subtracted and then P is scaled by 4 to become 12. Because the sign bit has weight 8 here, 12 is read as -4. But this is below the table's lower limit, at $D = 18/16$, of $P = -8/3 \cdot 18/16 = -3$. Hence a second subtraction is skipped and P is further scaled to -16, which is mistaken for 0. We have now lost the whole of P, which was 3. Simply subtracting 3 from P before these two cycles would have yielded the exact same sequence of choices of quotient digit. This therefore gives a clean characterization of the error in this boundary case: it is equivalent to subtracting 3, scaled by a suitable power of 2, from the original dividend. It follows that $y - (y/x)x$ will be 3 so scaled, to within the precision normally achieved by division.

For $D = 21/16$, $P = 8/3 \cdot 21/16 = 7/2$. This quantity should have been reduced to 7/8 then scaled to 7/2. Instead it is scaled to 14 and then mistaken for -2. This puts it within P's safe operating range. Hence the effect is the same as if we had started with $P = -1/2$. But this represents a loss of 4 from P. Hence $y - (y/x)x$ will be a power of two. Continuing these calculations for the remaining three boundary cases of D, we find a similar loss of 4 in every case.

A perturbation of D and P sufficiently small that the missing table entry is still hit will not be sufficient to change the gross loss of bits from the high end of the partial remainder. Hence $y - (y/x)x$ will continue to be 3 times a power of 2 when D is near 18/16 and a power of 2 for other D.

Table 1: Number of Pentium Errors, by precision of operands and quotient

Operand Precision	Quotient Precision			
	Single	Double	Extended	TOTAL
Single	1738	7863	9915	7.037×10^{13}
Double	5.009×10^{20}	2.266×10^{21}	2.858×10^{21}	2.028×10^{31}
Extended	2.101×10^{27}	9.502×10^{27}	1.199×10^{28}	8.507×10^{37}
ERROR RATE	2.470×10^{-11}	1.117×10^{-10}	1.409×10^{-10}	

4 Rate of Errors

The highly nonuniform distribution of erroneous operands makes it exceedingly difficult to predict the error rate of any given application. One approach is to describe the general characteristics of the bug and to let the user determine if possible how those characteristics interact with those of the application at hand.

We list here a table of the number of errors caused by the bug, as a function of the precision of the operands and the quotient. The operand precision affects mainly the number of operands defining the total population from which the errors are sampled.

The quotient precision affects the number of errors of a given magnitude. The 1738 single precision quotient errors are a subset of the 7863 double precision errors, and the latter are in turn a subset of the 9915 extended precision errors.

The errors turn out to be uniformly distributed with regard to the cycle on which they happen (and hence with regard to the logarithm (exponent) of the corresponding relative error), starting with the first cycle at which an error is possible, namely cycle 9, and continuing on to cycle 34, the last cycle of the Pentium division algorithm. The relative number of errors in each column of the above table can be roughly predicted given the number of bits of mantissa for each, namely 23, 52, and 63 (not counting the most significant bit in any of these cases). The following table classifies all 9915 errors tabulated above; we have clearly missed some 600 errors at cycles 33 and 34, whose small size makes them hard to distinguish from normal truncation error.

5 Random Computation

We may think of the uniform-distribution model as amounting to a program consisting of just the instruction FDIV, with random data. A natural counterpoint to this is a single datum, for simplicity the number 1, operated on with random instructions.

To explore this dual point of view we initialized a database to contain just the number 1. We then randomly added, subtracted, multiplied, and divided numbers, some from this database, some being more copies of the number 1, and put the results into the database. When the database reached its capacity of

Table 2: Classification of errors

	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	3149	1266	2357	1070	2073	9915

750,000 numbers, further incoming numbers displaced existing numbers removed from randomly selected locations in the database.

To prevent "database meltdown" through cumulative error, we kept track of how many times a number had been "put through the mill" and discarded numbers that had been operated on more than 100 times. More precisely, the depth of incoming 1's was taken to be 0, and the depth of constructed numbers was taken to be 1 plus the maximum depth of its operands. At depths of 200, cumulative errors on the order of 10^{-6} were produced and depth 300 resulted in "total meltdown." Limiting the depth to 100 largely avoided cumulative errors greater than 10^{-12}.

To simulate the typed discipline of a real database we assumed that the 1's were counts of kumquats, and distinguished kumquats, square kumquats, etc. Types were limited to kumquat^i for i ranging from -3 to 3, and operations that added kumquats to square kumquats or produced illegal types were discarded.

Two databases were maintained, clean and dirty. The difference was that divisions destined for the clean database were computed using the Mathisen-Moler bug workaround. Any discrepancies larger than 10^{-16} between the two databases were logged, giving the time (in cycles and ticks), clean and dirty values, relative error, depth, type, and cause of the discrepancy as a number. The first appearance of cause number n indicates the n-th FDIV error to occur during the computation. Subsequent appearances of n indicate relative errors larger than 10^{-16} resulting from operating on a number influenced by the n-th FDIV error.

The output from a typical run is given in Table 3.

The time is given in cycles consisting of 750,000 ticks, a tick being the production of one number passing all tests for membership in the database. The above table records all errors logged in the first 400 cycles. It is particularly noteworthy that no errors at all were logged during the second half of those 400 cycles. But then at cycle 402 the 13th FDIV error occurred, and half a cycle later combined with one of the offspring of the 5th error. A flurry of errors attributed to the 5th and 7th division errors then started up, and for the next thirty cycles a storm ensued reaching rates of up to ten errors per second. Gradually the storm calmed down, and at cycle 430, 1700 errors later, died out altogether. In all that time not a single fresh FDIV error had been logged, and the 14th FDIV error did not occur until cycle 512.

However a total of 12 FDIV errors had occurred during the 300 million ticks comprising the first 400 cycles, 50 million of which were divisions. This corresponds to an error rate of one in 4 million. This is a thousand times greater than the rate predicted by the uniform-distribution model.

Of greater concern is the tendency of errors to propagate through a database. This example graphically illustrates the chaotic nature of this propagation: errors may have one or two descendants over a very long period, but then errors that one assumes must have been flushed completely out of the system by now can suddenly reappear like the creature from Aliens and wreak havoc. When the 1700 indirectly caused errors are added to the 12 direct errors, the total error rate rises to one in 30,000 divisions!

Table 3: Output from a typical run

__TIME__	________DATA_______					
Cycle:Tick	Clean	Dirty	Error	Depth	Type	Cause
17:497981	0.16666668154	0.16666667905	1.5e-08	58	-1	1
17:616863	5.9999994647	5.9999995541	-1.5e-08	59	2	1
17:665562	-8561.0008917	-8561.0008917	-2.9e-13	59	-1	1
37:257439	-0.6666667262	-0.66666671626	1.5e-08	99	0	2
37:527542	-0.6666667262	-0.66666671626	1.5e-08	100	1	2
40:656817	0.83333330718	0.83333203561	1.5e-06	95	-2	3
41:702595	0.16666666668	0.16666666668	1.5e-11	98	-2	4
42:22138	0.16666666668	0.16666666668	1.5e-11	99	-3	4
42:116157	0.33333333336	0.33333333336	1.5e-11	100	0	4
44:669106	-0.33333333345	-0.33333333343	5.8e-11	90	0	5
45:135182	1.3082215759	1.3082215758	5.8e-11	99	3	5
45:181839	-1.3333332431	-1.3333332431	-1.5e-11	100	0	5
45:388059	1.3082215759	1.3082215758	5.8e-11	100	2	5
46:344255	-0.58333332942	-0.58333301153	5.4e-07	98	2	6
46:519308	-61017.845279	-61017.845279	-1.3e-15	100	3	5
46:740247	-0.58333332942	-0.58333301153	5.4e-07	99	0	6
47:138389	1.2215684139	1.2215684138	6.2e-11	100	3	5
47:275623	-0.497968866	-0.49796854811	6.4e-07	100	0	6
48:372879	-1.833333333	-1.8333332535	4.3e-08	99	0	7
51:277507	-0.66666690447	-0.66666686473	6e-08	92	1	8
94:659253	0.1666666669	0.16666666686	2.3e-10	98	-1	9
95:17753	0.1666666669	0.16666666686	2.3e-10	99	0	5
95:57593	-5.1666666669	-5.1666666669	7.5e-12	100	0	7
116:611163	-0.45833333333	-0.45833333333	6.6e-13	39	2	10
138:230655	-228378.66667	-228378.66667	7e-13	59	1	11
138:268063	-4.3786926e-06	-4.3786926e-06	-7e-13	60	0	7
138:502339	-228377.66667	-228377.66667	7e-13	60	1	7
138:521008	-4.3787118e-06	-4.3787118e-06	-7e-13	61	0	7
195:80147	3.6666666667	3.6666666667	6.6e-13	17	-1	12
195:103551	3.6666666667	3.6666666667	6.6e-13	18	0	12
195:114214	0.60604609328	0.60604609328	-6.6e-13	100	0	5
195:358813	-21.853931025	-21.853931025	6.6e-13	76	-2	7
196:231884	-22.853931025	-22.853931025	6.3e-13	77	-2	7

On the domain of small bruised integers the error rate rises several more orders of magnitude. Choose two integers from 1 to 100 with equal probability, subtract 10^{-6} from each, and divide one by the other. The probability of encountering a cycle 10 error (the second largest possible) is 0.08%, for a cycle 11 error it is 0.15%, and for cycle 12, 0.17%.

A good example of this is given by 4.999999/14.999999. This should be 0.33333329, but on the Pentium it turns out to be 0.333329, a cycle 10 error.

References

[Atk68] D.E. Atkins. Higher-radix division using estimates of the divisor and partial remainders. *IEEE Transactions on Computers*, C-17(10), October 1968.

[CMMP95] T. Coe, T. Mathisen, C. Moler, and V. Pratt. Computational aspects of the Pentium affair. *IEEE J. Computational Sci. and Eng.*, March 1995.

[Rob58] J.E. Robertson. A new class of digital division methods. *IRE Transactions on Electronic Computers*, EC-7:218–222, September 1958.

[SB94] H.P. Sharangpani and M.L. Barton. Statistical analysis of floating point flaw in the Pentium^{TM} processor (1994). Available on World-Wide Web as http://www.intel.com/product/pentium/white11/index.html, November 1994.

[Toc58] T.D. Tocher. Techniques of multiplication and division for binary computers. *Quarterly J. of Applied Math*, 2:364–384, 1958.