The Mathematics of the Pentium Division Bug

Alan Edelman*

Abstract

Despite all of the publicity surrounding the Pentium bug of 1994, the mathematical details of the bug are poorly understood. We discuss these details, supply a new proof of the Coe–Tang result that the at risk divisors have six consecutive ones in positions 5 through 10. Also we explain why the worst case absolute error for arguments in $[1, 2)$ is on the order of 1e-5.

1 Introduction

Quite properly, the risk to the population of Pentium users emerged as the most provocative issue in the discussion of the Intel Pentium flaw. Unfortunately, evaluating this risk may well be infeasible. Other well-publicized issues include Intel's public relations policy (eventually chips were replaced automatically by request), the story of how Nicely found the bug, how Coe succeeded in reverse engineering the algorithm based on "bad" numerators and denominators, and those ubiquitous Pentium jokes [3]. Vaughan Pratt [12], holds the record for the most extensive computational experiments and classifications of the bug.

We also wish to emphasize that, despite the jokes, the bug is far more subtle than many people realize. Andrew Wiles had a "bug" in the first public draft of his proof of Fermat's Last Theorem – his work was too important for anybody to laugh. The bug in the Pentium was an easy mistake to make, and a difficult one to catch. One way to catch it directly is Kahan's [10] SRT division tester. Kahan's tester chooses arguments more cleverly than random testing, increasing the likelihood of discovering whether an algorithm is somehow broken. His tester concentrates on and near the fenceposts, which is always a good place to look for difficulties. Randy Bryant [1] has a different sort of tester. He uses binary decision diagrams as a check of the validity of a PD table. This is the table in which the bug appears. His algorithm explicitly checks that partial remainders remain within the critical region. It is not immediately clear that this tester could be used directly on the Pentium chip.

*Department of Mathematics Room 2-380, and Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, edelman@math.mit.edu, supported by a fellowship from the Alfred P. Sloan Foundation

Part of what make the bug so interesting is that people can make unfortunate inferences. The most important wrong conclusion one might reach from this paper is that the SRT division algorithm is so complicated that it can not be tested and should not be used. The issue is emotional, akin to avoiding airplane travel after an accident. The analysis in this paper is not needed to test chips, but does reveal how interesting the Pentium bug truly is.

This article is a self-contained mathematical discussion of the bug, and only the bug. Sections 2 through 4 contain a complete mathematical specification of the algorithm which would be sufficient for the reader to try to see if he can devise buggy numerators and denominators without ever touching a Pentium chip.

We begin with a quick discussion of radix (or base) 4 SRT division, and its carry-save implementation. A very nice tutorial on the subject was recently written by David Goldberg [9]. We then give a new proof of the Coe–Tang [6] result that it takes six ones to reach a flaw. The first part of the proof begins with a simple analysis of the inequalities that either prevent you or allow you access to an erroneous table entry. The second part of the proof is an arithmetic puzzle not so very different in spirit from the sort found in recreational mathematics, where one has to replace letters with digits to make a correct sum as in this one from a collection by James Fixx [8]:

```
  S E N D        9 5 6 7
+ M O R E   =  + 1 0 8 5 ,
---------      ---------
M O N E Y      1 0 6 5 2
```

uniquely.

We conclude with an explanation of why a defective Pentium was always guaranteed to have an absolute error no bigger than `5e-5`. when it divided inputs are in the standard interval $[1, 2)$.

2 SRT Division

We now present the radix 4 version of the Sweeney, Robertson, and Tocher (SRT) division algorithm. This algorithm computes a radix 4 quotient where the digits are not from the set $\{0, 1, 2, 3\}$ as might be expected from base 4, but rather $\{-2, -1, 0, 1, 2\}$. The extra digit introduces a very useful "slack" into the computation. Binary multiplication by digits 0, ± 1, and ± 2 also require simpler hardware than would, say, the digit 3.

Letting p and d denote the numerator and the denominator, we may as well assume that $1 \leq p, d < 2$. The SRT division algorithm may be expressed succinctly in conceptual pseudocode:

RADIX 4 SRT DIVISION

$p_0 := p$

for $k = 0, 1, \ldots$

As a function of p_k and d, determine a digit $q_k \in \{-2, -1, 0, 1, 2\}$ in such a way that

$$p_{k+1} := 4(p_k - q_k d)$$

will satisfy $|p_{k+1}| \leq \frac{8}{3}d$

end

$p/d = \sum_{i=0}^{\infty} q_i/4^i$.

The determination of q_k is performed by looking up a table described in Section 4. How the lookup table works on the Pentium is not important yet. The trouble with a defective Pentium is, by accident, its table contains entries q_k which if accessed would produce p_{k+1} that would violate the requirement $|p_{k+1}| \leq \frac{8}{3}d$.

If we replace the set $\{-2, -1, 0, 1, 2\}$ with $\{0, \ldots, 9\}$, and the 4 with a 10, and finally the inequality $|p_{k+1}| \leq \frac{8}{3}d$ with $p_{k+1} \in [0, 10d)$, then the pseudocode describes ordinary base 10 long division.

3 Understanding why SRT works

It is easy to see that if $|p_k| \leq \frac{8}{3}d$ then at least one of the five values of $p_k - q_k d$ has absolute value $\leq \frac{2}{3}d$. Figure 1 (with unit length d) illustrates this. The indicated q_k value translates each interval to the $q_k \leftarrow 0$ interval. Overlaps between upper and lower intervals show where either of two choices for q_k will work. The $q_k \leftarrow 0$ interval, when scaled up by a factor of 4, remains within the interval $|p| \leq \frac{8}{3}d$; that is why we may run the algorithm ad infinitum.

We can therefore find a q_k for which

$$p_{k+1} = 4(p_k - q_k d) \tag{1}$$

has absolute value $\leq \frac{8}{3}d$, for every $k > 0$.

Since equation (1) is equivalent to

$$\frac{p_k}{d}4^{-k} = \frac{q_k}{4^k} + \frac{p_{k+1}}{d}4^{-(k+1)},$$

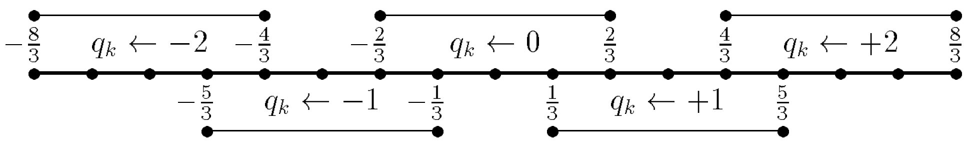

Figure 1: The above figure shows the interval $[-\frac{8}{3}d, \frac{8}{3}d]$. Each "tic" mark represents the value $d/3$. Five subintervals are shown: three labeled from above and two from below, with values for q_k. If p is in one of the five subintervals, and q_k is the digit labeling the subinterval, then the operation sending p to $p - q_k d$ sends the subinterval to the one corresponding to $q_k \leftarrow 0$. The operation sending p to $4(p - q_k d)$, then, keeps p from leaving the interval $[-\frac{8}{3}d, \frac{8}{3}d]$.

we may prove by induction that

$$\frac{p}{d} = \left(q_0 + \frac{q_1}{4} + \cdots + \frac{q_{k-1}}{4^{k-1}}\right) + \frac{p_k}{d}4^{-k}. \tag{2}$$

Letting $k \to \infty$ in Equation (2) proves that the SRT algorithm computes the correct quotient. $\square$

Notice that because of the overlap regions, the representation of the quotient in terms of the q_k is not uniquely determined, but the value of the quotient is, of course, uniquely determined.

There is a popular misconception that the SRT algorithm is capable of correcting for "mistakes" by using the redundant set of digits. Though the overlap in the regions may allow occasionally either of two choices for q_k, if an invalid q_k is chosen the algorithm will never recover.

A consequence of Equation (2) is that

$$p_k = d\sum_{i=k}^{\infty} q_i/4^{i-k} = d(q_k + \frac{q_{k+1}}{4} + \frac{q_{k+2}}{4^2} + \ldots).$$

Summing the geometric progression with the extreme choices of all $q_i = +2$ or all $q_i = -2$ shows that the requirement $|p_k| \leq \frac{8}{3}d$ is not an arbitrary choice, but a necessary ingredient in the algorithm.

On defective Pentiums, for certain values of d and for certain values of p_k just a little smaller than $\frac{8}{3}d$, the chip's table supplies the value 0 for q_k rather than 2. Then p_{k+1} turns out to be nearly $10d$, which is way outside of the range representable by our geometric progression. Consequently, after the Pentium has supplied $q_k = 0$ instead of 2, it cannot recover a correct quotient.

A small observation that we will use later is

Lemma 3.1 *If $q_k = 2$, then $p_{k+1} \leq p_k$.*

Proof This is an immediate consequence of the bound $p_k \leq \frac{8}{3}d$ and Equation (1). $\square$

4 Quotient digit selection on the Pentium

The Pentium uses a two-dimensional lookup table to obtain q_k. One entry is D, the binary number of the form `x.yyyy`, i.e. the integer multiple of 1/16, that approximates d from below by four bits. Mathematically,

$$D \leq d < D_+ \equiv D + \frac{1}{16}. \tag{3}$$

The other entry is P_k which is obtained from the carry-save representation to be explained momentarily. P_k is a binary number of the form `xxxx.yyy`, i.e. an integer multiple of 1/8. It is related to p_k by the condition that

$$P_k \leq p_k < P_k + \frac{1}{4}, \tag{4}$$

but this condition does not define P_k given p_k, it merely narrows it down to two choices. The choice that is made will be defined in Equation (6).

The lookup table [6, 13] that computes q_k as a function of P_k and D appears in Figure 4 below. The five colors indicate the five digits $+2, +1, 0, -1, -2$ from top to bottom. The five "stars" at the top of the table indicate table entries that must be 2 but erroneously return 0 on flawed Pentium chips. Note that the five columns are completed using the Coe model, while other columns have ambiguous entries. Indeed, we only know these values because of the bug; otherwise it would be impossible to know. The five entries on the bottom of the table with white borders are not reached hence any value for q_k may be placed there. Any proposed verifier of the PD table must be cognizant of the fact that these entries are unreachable or it may mistakenly determine the table invalid. As Pratt writes [12]

> 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.

The operative word is the word *believes* in the quotation above.

The five bad entries occur for those divisors d for which D has any of the five values 17/16, 20/16, 23/16, 26/16, 29/16. These are the values which allow us a peek at how the Pentium chip

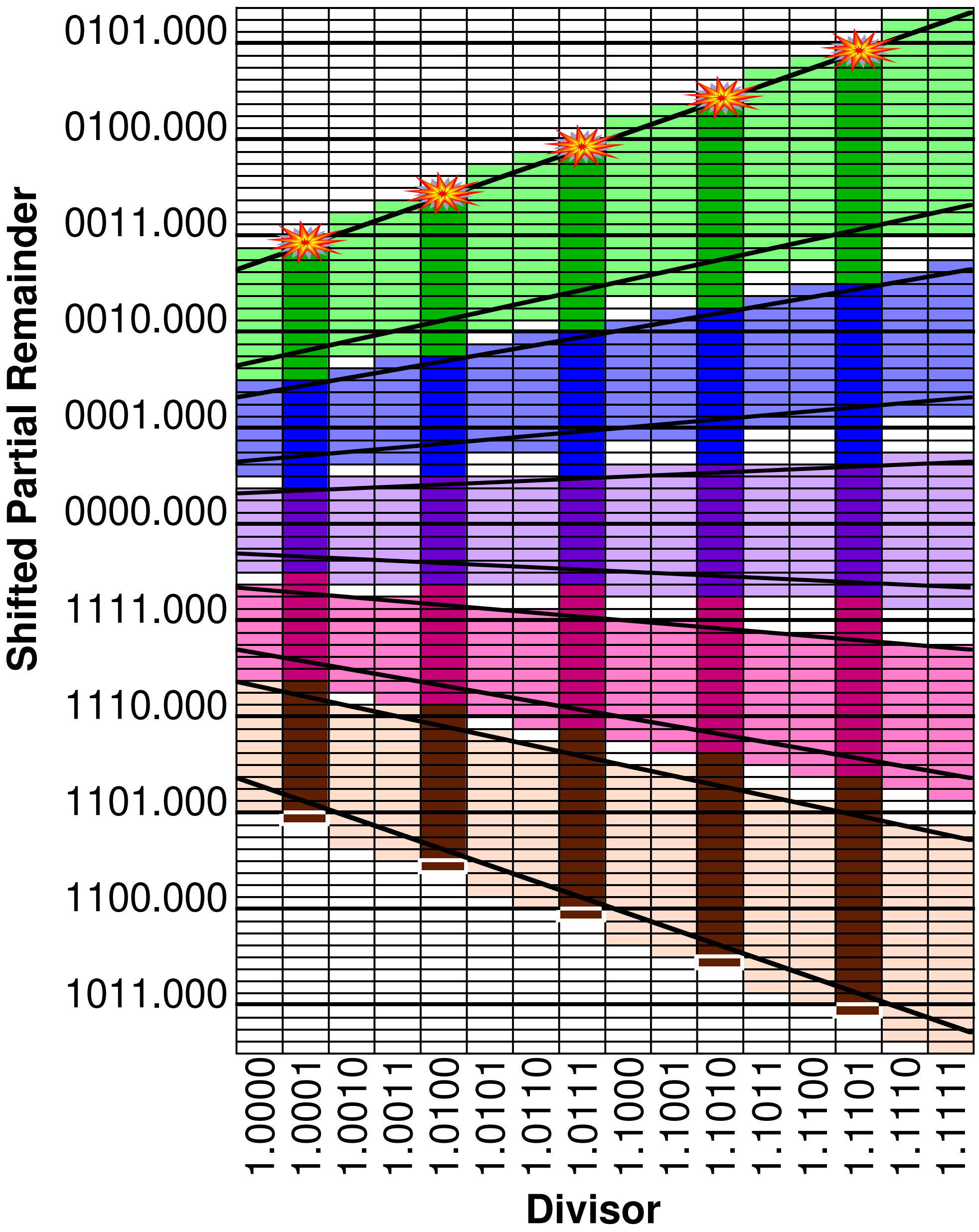

Figure 2: The quotient digit $q_k \in \{-2, -1, 0, 1, 2\}$ is a function of P_k and D. During one SRT division, D is fixed, so entries are accessed within one column. For example, if $1 \leq D < 17/16$, then only entries in the first column are accessed. The five trapezoidal bands (green, blue, etc.) indicate the digits q_k from $+2$ to -2 in decreasing order from the top. The five flawed entries that should be $+2$ but which instead are accidentally 0 are indicated by the explosion symbols at the top of the five columns. In these columns we know from the Tim Coe model the values of the entries. In other columns, there are empty squares in the interior where two possible entries are valid. Towards the top and bottom of the table are entries that a correctly working Pentium never accesses. Note the five entries towards the bottom without borders. They also are never accessed.

is performing division. For the other D values, there is no algorithm pathology, so there is no independent way to be sure which digits are selected in the overlap regions. Hence the white boxes in the interior of the lookup table indicating that two possible choices are equally correct. It is worth mentioning that a careless reading of [7] might suggest that the table is sign symmetric. The table is not sign symmetric, and one may speculate that this is one of several errors that contributed to the bug.

A matlab program to produce table entries follows. Half integers donote that any value is allowed while infinity denotes entries that are not accessed:

```
P=8*(-6:.125:6)';q=0*P;
for d=[1:(1/16) : 2-(1/16)];    D=floor(16*d); Dp=D+1;
qq= (P >= floor(-8*Dp/6)-1) + (P >= ceil(-5*D/6))  +  ...    %q=-2
    (P >= floor(-4*Dp/6)-1) + (P >= ceil(-2*D/6))  +  ...    %q=-1
    (P >= floor(  -Dp/6)-1) + (P >= ceil(   Dp/6)) +  ...    %q= 0
    (P >= floor( 2*D/6)-1)  + (P >= ceil( 4*Dp/6)) +  ...    %q= 1
    (P >= floor( 5*D/6)-1)  + (P >= ceil( 8*Dp/6)) ;         %q= 2
    q = [q qq];
end
q=(q-5)/2; q(abs(q)==2.5)=Inf*q(abs(q)==2.5); q(:,1)=P/8;q
```

For the remainder of this paper, we are only concerned with the five bad columns in the lookup table. There are the columns with the explosion symbols in Figure 4. Mathematically, they are the columns where $\frac{2}{3}D_+$ is an integer multiple of 1/8. Following Coe and Tang [6], a flawed entry exists in the box corresponding to the value

$$\text{buggy entry} = P_{\text{Bad}} = \frac{8}{3}D_+ - \frac{1}{8}.$$

The information expressed in colors in Figure 4 may be expressed succinctly with thresholds by identifying which of five intervals P_k falls in:

$$P_q^{\text{Min}} \leq P_k \leq P_q^{\text{Max}}. \tag{5}$$

The thresholds according to [6] are

q	P_q^{Min}	P_q^{Max}
-2	$-\frac{8}{3}D_+$	$-\frac{1}{4}-\frac{4}{3}D_+$
-1	$-\frac{1}{8}-\frac{4}{3}D_+$	$\lfloor -\frac{1}{4}-\frac{1}{3}D_+ \rfloor$
0	$\lfloor -\frac{1}{8}-\frac{1}{3}D_+ \rfloor$	$\lceil -\frac{1}{8}+\frac{1}{3}D_+ \rceil$
1	$\lceil \frac{1}{3}D_+ \rceil$	$-\frac{1}{8}+\frac{4}{3}D_+$
2	$\frac{4}{3}D_+$	$-\frac{1}{8}+\frac{8}{3}D_+$

,

where the floor and ceiling symbols round to multiples of 1/8.

It is easy to check that the q chosen in this manner satisfies the constraints of the algorithm specified in Section 2. We stress once again that the thresholds given in the table above are meant to apply only in the five buggy columns.

4.0.1 The computation of P_k: Carry-Save addition

Imagine adding 100 numbers on a calculator. On many calculators after typing $x_1 + x_2$ the sum would be displayed, then folding in x_3 the new sum is obtained, etc. On computers it is convenient to avoid the carry propagation by leaving the result in so-called "carry-save" format (See [2, pp.668–669]. In this format $x_1 + x_2$ is represented as $s_2 + c_2$. When we add in x_3 the result is represented as $s_3 + c_3$, etc. The s_i and c_i are known as the sum and carry word. The basic idea is that when computing the sum of $s_2 + c_2 + x_3$ in binary, every column can add up to $0, 1, 2,$ or 3 so the modulo 2 sum of the result (0 or 1) is stored in the sum word, and the carry bit is stored in the carry word.

Here is an example:

x_1	`0 1 0 1 1`
x_2	`0 1 1 0 1`
s_2	`0 0 1 1 0`
c_2	`1 0 0 1 0`
x_3	`0 1 1 1 0`
s_3	`1 1 0 1 0`
c_3	`0 1 1 0 0`

Generally speaking `sum(a,b,c)` is $a+b+c \bmod 2$ in mathematical language, and $a \oplus b \oplus c$ in a more computer science style language. `carry(a,b,c)` may be expressed as $a+b+c \geq 2$ in a matlab sort of language, or as $(a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$. The numbers c_k and s_k constitute the carry-save representation of $\sum_{i=1}^{k} x_k$. This representation is not unique. However, the least significant bit of c_k is always 0.

On the Pentium, p_k is represented in the carry-save form $c_k + s_k$ so that the expression $p_{k+1} =$

$p_k - q_k d$ is computed using a carry-save adder. If q_k is positive $-q_k d$ is represented as the one's complement of $q_k d$.[1] Since forming the two's complement is a two part process, it is worthwhile to simply complement the bits when forming $q_k d$, and delay the injection of the addition of 1 by changing the least significant bit of c_k to a one.

In general $p_k - q_k d$ is obtained from d by a combination of shifting and/or ones complementing or zeroing. Since these are fast operations on a computer, this explains why the digit set $\{-2, -1, 0, 1, 2\}$ is so useful. To perform addition correctly, if a one's complement number is used, a 1 is added in to the least significant carry bit before shifting. We will see this in the example.

Given that $p_k = c_k + s_k$, it is natural to define

$$P_k \equiv C_k + S_k, \tag{6}$$

where C_k and S_k represent c_k and s_k respectively rounded down to the nearest bit. Since

$$C_k \leq c_k < C_k + \frac{1}{8} \text{ and } S_k \leq s_k < S_k + \frac{1}{8},$$

we can conclude that $P_k \leq p_k + \frac{1}{4}$.

Radix 4 SRT Division with Carry-Save Addition Pentium Style

$D := \lfloor d \rfloor_{1/16}$

$S_0 := p_0, \quad C_0 := 0$

for $k = 0, 1, \ldots$

  $P_k := \lfloor c_k \rfloor_{1/8} + \lfloor s_k \rfloor_{1/8}$

  $q_k :=$ table-lookup(P_k, D)

  compute $(-q_k d)$ by zeroing, shifting, and/or ones complementing

  $c_{k+1} + s_{k+1} := 4(c_k + s_k + (-q_k d))$ carry-save addition and shift

    Correct for one's complement by injection into c_{k+1} if $q_k < 0$

end

$p/d = \sum_{i=0}^{\infty} q_i/4^i$.

The following example illustrates the division of 1.875 by 1.000 using fewer bits than is actually used in the Pentium. (The actual number of bits used in the Pentium is important (see [12, Section

[1] One's complement represents negative integers by complementing every bit. Two's complement is obtained by complementing a bit string and then adding one. Mathematically, a two's complement number is correct modulo a power of 2.

2, last paragraph]), but not for our concerns here.) Of course, dividing by 1 is trivial, but the important features of the algorithm are illustrated below.

1.875=	`0001.111 00000000000`	s_0=1.875 in binary
	`0000.000 00000000000`	c_0 is initially set to 0
-2x1=	`1101.111 11111111111`	2 (from lookup table, where $P_0 = C_0 + S_0$) in one's complement
	`0000.011 11111111100`	s_1 is the sum from above with the two place shift
	`1111.000 00000000100`	c_1 is the carry, with shift, and one's complement correction bit
-(-1)x1=	`0001.000 00000000000`	$S_1 = .011 = 3/16$, $C_1 = 1111.000 = -1$, $P_1 = -13/16$.
	`1001.111 11111100000`	
	`1000.000 00000100000`	no correction bit since no one's complement in previous iteration
-2x1 =	`1101.111 11111111111`	
	`0000.000 00011111100`	
	`1111.111 11100000100`	
-0x1=	`0000.000 00000000000`	
	`1111.111 11111100000`	
	`0000.000 00000100000`	
	`0000.000 00000000000`	
	`1111.111 11100000000`	
	`0000.000 00100000000`	

We have thus computed the representation $1.875 = 2 - \frac{1}{4} + \frac{2}{16} + \frac{0}{32} + \dots$.

5 Analyzing the bug

5.1 It is not easy to reach the buggy entry

We proceed to prove a lemma which states that the buggy entry P_{Bad} can only be reached from the entry below it in the PD table. This entry just below the bad entry plays the role of a foothold. The existence of this foothold is a subtle phenomenon that may not have been readily guessed from general properties of SRT division. I believe that its existence has surprised everybody. Any thought that each table entry is somehow equally likely to be reached is very wrong.

The table entries that play an important role in reaching the bug are

Description	Symbol	Value
Buggy Entry	P_{Bad}	$-\frac{1}{8}+\frac{8}{3}D_+$
Foothold	$P_{\text{Bad}}-\frac{1}{8}$	$-\frac{1}{4}+\frac{8}{3}D_+$
Top of $q=-2$ region	P_{-2}^{Max}	$-\frac{1}{4}-\frac{4}{3}D_+$
Top of $q=-1$ region	P_{-1}^{Max}	$-\frac{1}{4}-\frac{1}{3}D_+$

The result that we proceed to prove in Lemma 5.1 below is that one can only reach the buggy entry from the foothold. We will also show that it is possible to reach the foothold in four ways: from the top of the $q=-2$ region, the top of the $q=-1$ region, from the foothold itself or the entry below the foothold. However, if the foothold is reached from the latter two of the four routes, the buggy entry can not be reached on the next step. Therefore, there are only two viable routes to reach the bug. It is worth emphasizing that reaching the foothold is necessary for reaching the buggy entry, but not sufficient. It is quite possible to reach the foothold, but not hit the buggy entry on the next step.

We now proceed with our analysis. Directly from the definition of D_+ given in Equation 3, we can write the following identities for $q_k D_+$ in terms of the binary expansion of $d = 1.d_1d_2d_3d_4d_5\ldots$:

$$\begin{array}{rcll} 2D_+ &=& 0\,0\,1\,d_1.\,d_2\,d_3\,d_4\;0 & +\frac{1}{8} \\ D_+ &=& 0\,0\,0\;1.\;d_1\,d_2\,d_3\,d_4 & +\frac{1}{16} \\ 0D_+ &=& 0\,0\,0\;0.\;\;0\;\,0\;\,0\;\,0 & , \\ -D_+ &=& 1\,1\,1\;0.\;\bar{d}_1\,\bar{d}_2\,\bar{d}_3\,\bar{d}_4 & \\ -2D_+ &=& 1\,1\,0\,\bar{d}_1.\,\bar{d}_2\,\bar{d}_3\,\bar{d}_4\;0 & \end{array} \tag{7}$$

where the negatives numbers are expressed in two's complement notation.[2] The chopping is a round down process, but D_+ is akin to a round up, hence the existence of the $\frac{1}{8}$ and $\frac{1}{16}$ terms.

Therefore in terms of P_k and D_+, we have the version of Equation (1) that is actually computed on the Pentium:

$$P_{k+1} = 4(P_k - q_k D_+) + R_k, \tag{8}$$

where R_k is determined by a few higher order bits. Working out the exact value of R_k requires

[2]Two's complement represents integers by reducing modulo a power of 2. Add D_+ and $-D_+$ or add $2D_+$ and $-2D_+$ (and multiply by 16 to remove the binary point) to see that the sum is a power of 2.

careful attention to the algorithmic details. The reader may verify that

$$R_k = 0.0s_4s_5 + 0.0c_4c_5 + \left\{ \begin{array}{l} 0.0\ d_5\, d_6 \\ 0.0\ 0\ \ d_5 \\ 0 \\ 0.0\ 0\ \ \bar{d}_5 \\ 0.0\ \bar{d}_5\, \bar{d}_6 \end{array} \right\} + \frac{1}{8} \times \text{carry-over}(s_6, c_6, \left\{ \begin{array}{c} d_7 \\ d_6 \\ 0 \\ \bar{d}_6 \\ \bar{d}_7 \end{array} \right\}) + \left\{ \begin{array}{c} -1/2 \\ -1/4 \\ 0 \\ 0 \\ 0 \end{array} \right\}, \text{ if } \left\{ \begin{array}{c} q=-2 \\ q=-1 \\ q=0 \\ q=1 \\ q=2 \end{array} \right.$$

where the s_i, c_i, and d_i are the appropriate bits in the sum, carry, and divisor respectively.[3] The function carry-over(b_1, b_2, b_3) is 1 if at least two of its arguments are 1. The last correction term of $-1/2$ and $-1/4$ is a direct consequence of Equation (7).

Taking the maximum value of the parameters we see that

$$R_k \leq R_k^{\text{Max}} \equiv \left\{ \begin{array}{c} 3/4 \\ 3/4 \\ 7/8 \\ 1 \\ 5/4 \end{array} \right\}, \quad \text{if} \quad \left\{ \begin{array}{c} q=-2 \\ q=-1 \\ q=0 \\ q=1 \\ q=2 \end{array} \right. . \tag{9}$$

We assume that D_+ is a multiple of $3/16$ i.e., it corresponds to one of the five flawed table entries. For each value of q_k, we may tabulate the largest possible P_{k+1} that may be reached as a function of the extreme P_k by using equations (8) and (9). Therefore the table below is a transition table that shows how high one can reach in the table given the previous value of q_k. This is our first glimpse at just how difficult it is to reach the bad entry P_{Bad}.

	If P_k is at most $\downarrow$,	then P_{k+1} is at most $\downarrow$	can reach the foothold
q_k	P_k	$P_{k+1}^{\text{Max}} = 4(P_k - q_k D_+) + R_k^{\text{Max}}$	
-2	$P_{-2}^{\text{Max}} = -\frac{4}{3}D_+ - \frac{1}{4}$	$\frac{8}{3}D_+ - \frac{1}{4} = P_{\text{Bad}} - \frac{1}{8}$	yes
-1	$P_{-1}^{\text{Max}} \leq -\frac{1}{3}D_+ - \frac{1}{4}$	$\frac{8}{3}D_+ - \frac{1}{4} = P_{\text{Bad}} - \frac{1}{8}$	yes
0	$P_0^{\text{Max}} \leq \frac{1}{3}D_+$	$\frac{4}{3}D_+ + \frac{7}{8} < P_{\text{Bad}} - \frac{1}{8}$	no
1	$P_1^{\text{Max}} = \frac{4}{3}D_+ - \frac{1}{8}$	$\frac{4}{3}D_+ + \frac{1}{2} < P_{\text{Bad}} - \frac{1}{8}$	no
	If P_k is at most $\downarrow$,	then P_{k+1} is at most $\downarrow$	reach buggy entry
2	$P_{\text{Bad}} - \frac{1}{8} = \frac{8}{3}D_+ - \frac{1}{4}$	$\frac{8}{3}D_+ + \frac{1}{4} > P_{\text{Bad}}$	yes
2	$P_{\text{Bad}} - \frac{1}{4} = \frac{8}{3}D_+ - \frac{3}{8}$	$\frac{8}{3}D_+ - \frac{1}{4} = P_{\text{Bad}} - \frac{1}{8}$	no

(10)

[3] We regret the abuse of notation – everywhere else in this paper s_i refers to all of the bits in the sum word of the ith iteration, but here it represents the ith bit during an unspecified iteration.

The two "less than" inequalities in the table above are a simple consequence of $D_{+} > 1$. In terms of Figure 4, the above table states that it is impossible to reach the explosion symbol even if you are at the very top of the brown, pink, purple, or blue regions. The only way to reach the explosion symbol is from the green entry immediately below it. Therefore, we denote this entry the foothold.

Lemma 5.1 *The sequence of P's and corresponding q's that lead to the bad entry are given below:*

$$\begin{array}{llclcl} P: & P_{-2}^{Max}/P_{-1}^{Max} & \longrightarrow & \left\{P_{Bad}-\frac{1}{8}\right\}_{m\geq 1} & \longrightarrow & P_{Bad} \\ q: & -2/-1 & \longrightarrow & \{2\}_{m\geq 1} & \longrightarrow & \text{(bad entry)}\ \cdot \\ R: & & R=R^{\max} & & R=R^{\max}-\frac{3}{8} & \end{array} \tag{11}$$

Here the subscript m in the middle column denotes a repetition of the entry $m \geq 1$ times, and the first column indicates that either a P_{-2}^{Max} or P_{-1}^{Max} may start the path to the bug corresponding to either $q = -2$ or $q = -1$ respectively.

Proof Our proof goes from right to left. The table in (10) shows that we can only reach P_{Bad} from $P_{\text{Bad}} - \frac{1}{8}$. The table also shows that for $q = -2$ or $q = -1$, we can just barely reach $P_{\text{Bad}} - \frac{1}{8}$, but only when $P_k = P_k^{\text{Max}}$ and $R_k = R_k^{\text{Max}}$. $P_{\text{Bad}} - \frac{1}{8}$ may also be reached from $P_{\text{Bad}} - \frac{1}{4}$, but if $P_k = P_{\text{Bad}} - \frac{1}{4}$ then $P_{\text{Bad}} - \frac{1}{4} \leq p_k < P_{\text{Bad}}$, and Lemma 3.1 guarantees that while the sequence produces 2's all future p_k will also satisfy $p_k < P_{\text{Bad}}$ which precludes reaching the flawed entry, i.e., reaching k with $P_k = P_{\text{Bad}}$. □

6 The "Send More Money" Puzzle for the Pentium

Theorem 6.1 (Coe,Tang) *A flawed table entry can only be reached if the divisor $d = d_1d_2d_3\ldots$ has the property that the six consecutive bits from d_5 to d_{10} are all ones. (Of course $1.d_1d_2d_3d_4$ must be one of the five bad values.)*

Proof

First assume that $m = 1$. (We later show in Lemma 6.1 that this is the only possibility.) The table below illustrates the carry-save division calculation that leads to the buggy entry. We devised a subscript notation to facilitate following the progression of fill-in. Our five main threads of argument are denoted by $\alpha, \beta, \gamma, \delta$ and ϵ. If you fill in the digits of our "send-more-money" in the following sequence, you will fill in the numbers in order:

$$\alpha_\spadesuit, \alpha_1$$
$$\beta_\spadesuit, \beta_1,$$
$$\gamma_\spadesuit, \gamma_1, \gamma_2$$
$$\delta_\spadesuit, \delta_1, \delta_2, \delta_3$$
$$\epsilon_\spadesuit, \epsilon_1, \epsilon_2$$

Only five observations are needed for this argument. They are associated with the $\spadesuit$ symbol, as with $\alpha_\spadesuit$. Then in sequence, we let $\alpha_1, \alpha_2, \ldots$ denote bits we can fill in more or less mechanically based on the carry-save division process. The only ideas used to fill in the mechanical entries are the shifted carry-sum division process and the possible shifting or complementing of the bits in d. Asterisks ($*$) or blank spaces indicate entries whose values are not of interest. Also of no interest are values to the left of the binary point.

Case I: Reaching the bug from -2

	s_{j-2}	$.*$	$*$	$*$	$1_{\alpha_\spadesuit}$	$1_{\alpha_\spadesuit}$	1_{γ_1}	1_{δ_2}	1_{ϵ_2}
$\boxed{q=-2}$	c_{j-2}	$.*$	$*$	$*$	$1_{\alpha_\spadesuit}$	$1_{\alpha_\spadesuit}$	1_{γ_1}	1_{δ_2}	1_{ϵ_2}
	$2d$	$.d_2$	d_3	d_4	$1_{\alpha_\spadesuit}$	$1_{\alpha_\spadesuit}$	1_{γ_1}	1_{δ_2}	1_{ϵ_2}
	s_{j-1}	$.*$	1_{α_1}	1_{α_1}	$1_{\gamma_\spadesuit}$	1_{δ_1}	1_{ϵ_1}		
$\boxed{q=2}$	c_{j-1}	$.1_{\alpha_1}$	1_{α_1}	$1_{\beta_\spadesuit}$	$1_{\gamma_\spadesuit}$	1_{δ_1}	1_{ϵ_1}		
	$-2d$	$.\bar{d}_2$	$\bar{d}_3$	$\bar{d}_4$	0_{α_1}	0_{α_1}	0_{γ_2}		
	s_j	$.*$	0_{γ_1}	0_{δ_2}					
$\boxed{\text{bug}}$	c_j	$.*$	$1_{\delta_\spadesuit}$	$1_{\epsilon_\spadesuit}$					

Case II: Reaching the bug from -1

s_{j-2}	$.*$	$*$	$*$	$1_{\alpha_\spadesuit}$	$1_{\alpha_\spadesuit}$	1_{γ_1}	1_{δ_2}	1_{ϵ_2}	
c_{j-2}	$.*$	$*$	$*$	$1_{\alpha_\spadesuit}$	$1_{\alpha_\spadesuit}$	1_{γ_1}	1_{δ_2}	1_{ϵ_2}	$\boxed{q=-1}$
d	$.d_1$	d_2	d_3	$1_{\alpha_\spadesuit}$	$1_{\alpha_\spadesuit}$	1_{γ_1}	1_{δ_2}	1_{ϵ_2}	
s_{j-1}	$.*$	1_{α_1}	1_{α_1}	$1_{\gamma_\spadesuit}$	1_{δ_1}	1_{ϵ_1}			
c_{j-1}	$.1_{\alpha_1}$	1_{α_1}	$1_{\beta_\spadesuit}$	$1_{\gamma_\spadesuit}$	1_{δ_1}	1_{ϵ_1}			$\boxed{q=2}$
$-2d$	$.\bar{d}_2$	$\bar{d}_3$	$\bar{d}_4$	0_{α_1}	0_{γ_2}	0_{δ_3}			
s_j	$.*$	0_{γ_1}	0_{δ_2}						
c_j	$.*$	$1_{\delta_\spadesuit}$	$1_{\epsilon_\spadesuit}$						$\boxed{\text{bug}}$

Explanation:

$\alpha_\spadesuit$ Since R (Lemma 5.1) must equal R^{max} we must have c_4, c_5, s_4, s_5, and the two d bits equal to 1.

$\beta_\spadesuit$ $8(P_{\text{Bad}} - \frac{1}{8})$ is even. (It can be 22, 26, 30, 34, or 38.) To obtain an even number, we must add a 1 to the 1_{α_1} immediately above.

$\gamma_\spadesuit$ $R = R^{\max} - \frac{3}{8}$ (Lemma 5.1). The 0_{α_1} immediately below reduces R by 2/8, and a 0 in the $\gamma_\spadesuit$ position would reduce R by another 2/8, giving 4/8.

$\delta_\spadesuit$ $8P_{\text{Bad}} \in \{23, 27, 31, 35, 39\}$ is 3 (mod 4).

$\epsilon_\spadesuit$ $8P_{\text{Bad}} \in \{23, 27, 31, 35, 39\}$ is 3 (mod 4).

The conclusion that we have so far is that if we begin with $q = -2$, we must then have that d_5, d_6, d_7, d_8, d_9 are all 1. If we begin with $q = -1$, then we conclude that d_5, d_6, d_7, d_8 are all 1.

We are almost, but not quite finished. Having all those ones at the $q = -2$ or $q = -1$ step allows us to go back yet another step. A quick analysis shows that the line above s_{j-2} can only be d or $2d$ and that it too has many ones. We can then guarantee that d_9 and d_{10} are also 1. $\square$

Lemma 6.1 *The value for m in Lemma 5.1 must be one.*

Proof We proceed with the same sort of send-more-money puzzle. The assumption of more than one $q = 2$ in sequence yields a contradiction. We encourage readers to try to find the contradiction themselves rather than reading the proof to follow. In the diagram below we do not use subscripts for values that we have already obtained from α, β, and γ in Theorem 6.1. We then continue the numbering convention with ζ.

Case I: Not reaching the bug from -2, 2, 2 Case II: Not reaching the bug from -1, 2, 2

Sequence leading to contradiction Sequence leading to branch

	s_{j-2}	$.*$	$*$	$*$	1	1	1	1_{ζ_5}	s_{j-2}	$.*$	$*$	$*$	1	1	1	
$\boxed{q=-2}$	c_{j-2}	$.*$	$*$	$*$	1	1	1	1_{ζ_5}	c_{j-2}	$.*$	$*$	$*$	1	1	1	$\boxed{q=-1}$
	$2d$	$.d_2$	d_3	d_4	1	1	1	1_{ζ_5}	d	$.d_1$	d_2	d_3	1	1	1	
	s_{j-1}	$.*$	1	1	1	1_{ζ_4}	$*$	1_{ζ_7}	s_{j-1}	$.*$	1	1	1			
$\boxed{q=2}$	c_{j-1}	$.1$	1	1	1	1_{ζ_4}	$*$	1_{ζ_7}	c_{j-1}	$.1$	1	1	1			$\boxed{q=2}$
	$-2d$	$.\bar{d}_2$	$\bar{d}_3$	$\bar{d}_4$	0	0	0	0_{ζ_6}	$-2d$	$.\bar{d}_2$	$\bar{d}_3$	$\bar{d}_4$	0	0		
	s_j	$.*$	0	0_{ζ_2}	$1_{\zeta_\spadesuit}$	0_{ζ_7}			s_j	$.*$	0	$\spadesuit$	1			
$\boxed{q=2}$	c_j	$.*$	1_{ζ_3}	0_{ζ_1}	$1_{\zeta_\spadesuit}$	$1_{\eta_\spadesuit}$			c_j	$.*$	$\spadesuit$	$\spadesuit$	1			$\boxed{q=2}$
	$-2d$	$.\bar{d}_2$	$\bar{d}_3$	$\bar{d}_4$	0	0	0		$-2d$	$.\bar{d}_2$	$\bar{d}_3$	$\bar{d}_4$	0	0		
	s_{j+1}			1_{η_1}	$0_{\theta_\spadesuit}$											
	c_{j+1}			1_{η_2}												

Explanation (Case I):

$\zeta\spadesuit$	If the next entry is the bug, $R = R^{max} - \frac{3}{8}$, otherwise $R = R^{max} - \frac{4}{8}$. Either way we need these two ones or we lose $\frac{5}{8}$.
$\eta\spadesuit$	We have already lost $\frac{4}{8}$ so the next digit is a 2 and we can not lose any more.
$\theta\spadesuit$	This is the contradiction! It must be a 0 because, if it were 1, then we would have a 1 in the line above where we already have a 0. On the other hand, to reach either the foothold or the buggy entry on the next step, the value must be 1 or R is too small. This is the contradiction.

Explanation (Case II):

Case II has only been started above. The only two possibilities for the three spades pattern in the diagram above are

$$\begin{matrix} 0 & \spadesuit \\ \spadesuit & \spadesuit \end{matrix} = \begin{matrix} 0 & 1 \\ 0 & 1 \end{matrix} \quad \text{and} \quad \begin{matrix} 0 & 0 \\ 1 & 0 \end{matrix},$$

since the foothold must be 2 modulo 4. Following through on the first possibility it becomes clear that the pattern

$$\begin{array}{l} \hline .{*}\,{*}\,1\,1\,0 \\ .{*}\,0\,1 \\ .{*}\,{*}\,{*}\,0\,0\,1\,0 \\ \hline \end{array}$$

perpetuates ad infinitum. Following the bug backwards in the second possibility would show that the pattern of bits for every step when $q = 2$ from the jth iteration until the bug is hit would have the pattern

$$\begin{array}{l} \hline .{*}\,0\,0\,1\,1 \quad 1 \\ .{*}\,1\,0\,1\,1 \quad 1 \\ .{*}\,{*}\,{*}\,0\,0\,0\,1 \\ \hline \end{array},$$

which quickly leads to a contradiction. $\square$

7 At least nine steps to failure

Naively, one might have thought that the bad table entries would be hit uniformly at random. If this had been the case, the bug surely have been noticed earlier and would have had the potential to cause more significant damage every instance it went unnoticed.

Even in the worst case of the error, the absolute error in the quotient is roughly $0.00005 = 5e-5$ when dividing two numbers in the interval $1 \leq x < 2$. This is the worst case, the actual errors can be far smaller. Vaughan Pratt [11] has exhaustively computed all the single precision errors and has found that the quotient 14909255/11009918 has an absolute error that is roughly $5e-5$. (In fact it is $4.65e-5$.) This may be used to test a Pentium chip. The correct answer is 1.35416585. One must exercise care in testing. Matlab 4.2 will trap and patch the bug. Other programs may simulate division in software, or use non-IEEE compliant formulas such as `x/y = (1/y)*x`. The highest relative error Pratt reported is roughly $6e-5$.

The reason the error is bounded is that, no matter what, the Pentium is guaranteed to compute q_0 through q_7 correctly. This is what we will now prove. The result is a straightforward consequence of the results in the previous section.

In particular, if the bug occurs at step 8, then we saw in Section 6 that step 6 must have the form shown in the diagram below which shows the sum, carry, and $q_k d$ bits respectively starting with bit number 4. Taking into account that all the carry bits are 0 at step 1, it is fairly easy to show (following and up and down "wave" motion) that the bit patterns must fall as in the diagram below. However, since q_1 is positive, it must follows that all of $q_2, \ldots, q_6$ are positive by observing the overlap in the $q_k d$ row from one step to the next. However, we known that $q_6 < 0$ so we have a contradiction. Therefore the pattern shown below as Step 6, which must occur to trigger the bug, can not appear before the seventh step, and hence the bug appears no earlier than the ninth step.

KEY TO DIAGRAM BELOW
START WITH $\spadesuit_0$ AND FOLLOW $\spadesuit_1$ UP THE CHART. THEN START WITH $\heartsuit_0$ AND FOLLOW THE $\heartsuit$S DOWN THE CHART. NEXT FOLLOW THE $\clubsuit$, THE $\diamondsuit$, THEN THE α AND THE β. THIS NOTATION MAKES IT EASIER TO SPOT THE FLOW ON THE PAGE.

	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	$1_{\spadesuit_5}$	$1_{\heartsuit_0}$	$1_{\clubsuit_0}$	$1_{\diamondsuit_0}$	1_{α_0}
Step 1:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	$1_{\spadesuit_5}$	$1_{\heartsuit_0}$	$1_{\clubsuit_0}$	$1_{\diamondsuit_0}$	1_{α_0}
	·	·	·	·	·	·	·	·	·	·	·	·	$1_{\spadesuit_4}$	$0_{\heartsuit_1}$	$0_{\clubsuit_1}$	$0_{\diamondsuit_1}$	0_{α_1}	0_{β_1}	·	·
Step 2:	·	·	·	·	·	·	·	·	·	·	·	·	$1_{\spadesuit_4}$	$1_{\heartsuit_1}$	$1_{\clubsuit_1}$	$1_{\diamondsuit_1}$	1_{α_1}	·	·	·
	·	·	·	·	·	·	·	·	·	·	·	·	$1_{\spadesuit_4}$	$1_{\heartsuit_1}$	$1_{\clubsuit_1}$	$1_{\diamondsuit_1}$	1_{α_1}	·	·	·
	·	·	·	·	·	·	·	·	·	$1_{\spadesuit_3}$	$1_{\spadesuit_3}$	$0_{\heartsuit_2}$	$0_{\clubsuit_2}$	$0_{\diamondsuit_2}$	0_{α_2}	·	·	·	·	·
Step 3:	·	·	·	·	·	·	·	·	·	$1_{\spadesuit_3}$	$1_{\spadesuit_3}$	$1_{\heartsuit_2}$	$1_{\clubsuit_2}$	$1_{\diamondsuit_2}$	·	·	·	·	·	·
	·	·	·	·	·	·	·	·	·	$1_{\spadesuit_3}$	$1_{\spadesuit_3}$	$1_{\heartsuit_2}$	$1_{\clubsuit_2}$	$1_{\diamondsuit_2}$	·	·	·	·	·	·
	·	·	·	·	·	·	$1_{\spadesuit_2}$	$1_{\spadesuit_2}$	$1_{\spadesuit_2}$	$0_{\heartsuit_3}$	$0_{\clubsuit_3}$	$0_{\diamondsuit_3}$	·	·	·	·	·	·	·	·
Step 4:	·	·	·	·	·	·	$1_{\spadesuit_2}$	$1_{\spadesuit_2}$	$1_{\spadesuit_2}$	$1_{\heartsuit_3}$	$1_{\clubsuit_3}$	·	·	·	·	·	·	·	·	·
	·	·	·	·	·	·	$1_{\spadesuit_2}$	$1_{\spadesuit_2}$	$1_{\spadesuit_2}$	$1_{\heartsuit_3}$	$1_{\clubsuit_3}$	·	·	·	·	·	·	·	·	·
	·	·	·	$1_{\spadesuit_1}$	$1_{\spadesuit_1}$	$1_{\spadesuit_1}$	$1_{\spadesuit_1}$	$0_{\heartsuit_4}$	$0_{\clubsuit_4}$	·	·	·	·	·	·	·	·	·	·	·
Step 5:	·	·	·	$1_{\spadesuit_1}$	$1_{\spadesuit_1}$	$1_{\spadesuit_1}$	$1_{\spadesuit_1}$	$1_{\heartsuit_4}$	·	·	·	·	·	·	·	·	·	·	·	·
	·	·	·	$1_{\spadesuit_1}$	$1_{\spadesuit_1}$	$1_{\spadesuit_1}$	$1_{\spadesuit_1}$	$1_{\heartsuit_4}$	·	·	·	·	·	·	·	·	·	·	·	·
	$1_{\spadesuit_0}$	$1_{\spadesuit_0}$	$1_{\spadesuit_0}$	$1_{\spadesuit_0}$	$1_{\spadesuit_0}$	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
Step 6:	$1_{\spadesuit_0}$	$1_{\spadesuit_0}$	$1_{\spadesuit_0}$	$1_{\spadesuit_0}$	$1_{\spadesuit_0}$	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
	·	$1_{\spadesuit_0}$	$1_{\spadesuit_0}$	$1_{\spadesuit_0}$	$1_{\spadesuit_0}$	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
	$\uparrow$ Bit #4																			

8 Conclusions

We have provided a new proof of the Coe-Tang result that the only divisors at risk are those divisors with six consecutive ones in the fifth through tenth positions. We also explain why the absolute error is bounded by `5e-5`, when dividing two numbers in the standard interval $[1, 2)$. Though we do not know exactly how the error occurred, we do point out that the error was probably more than an elementary careless accident. It is this author's position that the accident was more sophisticated, and perhaps we should think twice before laughing about the error at Intel's expense. In any event, it should be noted that the bug was discovered in 1994, and chips with copyright dates with 1995 or later should be non-defective.

9 Acknowledgement

I owe a great deal to Tim Coe for explaining carefully and patiently the reasoning behind his reverse engineering of the Pentium chip. This paper began life as an extended referee's report for the original version of the Coe, Tang paper [6]. Without their paper and Coe's explanation of the Pentium chip, this work would have been impossible. I also thank Velvel Kahan for many interesting discussions during January of 1995, while I was visiting Berkeley, and Vaughan Pratt for reviewing early drafts of this during June of 1995 while I was visiting Stanford. Finally, I thank Teddy Slottow for creating the beautiful color figure.

References

[1] Randall Bryant, Bit-level analysis of an SRT circuit, CMU technical report CS-140.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1992.

[3] At time of writing, one collection may be found on

`http://www-pcd.stanford.edu/cousins/pentium.html`

[4] Tim Coe, Inside the Pentium FDIV bug, *Dr. Dobb's Journal 20* (April, 1995), pp 129–135.

[5] Tim Coe, Terje Mathisen, Cleve Moler, and Vaughan Pratt, Computational aspects of the Pentium affair, *IEEE Computational Science & Engineering 2* (Spring 1995), pp 18–31.

[6] Tim Coe and Ping Tak Peter Tang, It takes six ones to reach a flaw, preprint.

[7] Jan Fandrianto, Algorithm for High Speed Shared Radix 4 Division and Radix 4 Square Root, *Proceedings of the 8th Symposium on Computer Arithmetic*, (1987), 73–79.

[8] James Fixx, Games for the Superintelligent. Doubleday and Company, Inc., Garden City, New York, 1972.

[9] David Goldberg, Tutorial on Computer Division, preprint.

[10] William Kahan, A Test for SRT Division, `http://HTTP.CS.Berkeley.EDU/~wkahan/srtest/srtest`, March 1995.

[11] Vaughan Pratt, *personal communication*, June, 1995.

[12] Vaughan Pratt, Anatomy of the Pentium Bug, *TAPSOFT'95, LNCS 915*, Springer-Verlag, Aarhus, Denmark, (1995), 97–107. `ftp://boole.stanford.edu/pub/FDIV/anapent.ps.gz`.

[13] H.P. Sharangpani and M.L. Barton, Statistical analysis of floating point flaw in the Pentium ^{TM} Processor (1994). `http://www.intel.com/product/pentium/white11.ps`