

Multiplication, Division

Prof. Loh

CS3220 - Processor Design - Spring 2005

February 24, 2005

1 Multiplication

Multiplication by hand in base 10 involves repeated multiplication of one number by the digits of the second number, and then adding all of the results together. In base 2, multiplication by a single bit is simplified by the fact that a bit only has two values (both of which are trivial to multiply by!). Multiplying an n -bit number by a single bit simply involves n AND gates, as illustrated in Figure 1. To multiply two n -bit numbers together, we simply need to perform n different $n \times 1$ -bit multiplications in parallel, shift the partial results properly, and add them all together. This is illustrated in Figure 2. The total gate delay is $O(1)$ for the 1-bit multiplies, zero for shifting (each shift is by a constant amount, so only wires are involved), and $O(\lg(n + \lg n)) \approx O(\lg n)$ gate delays for adding together n different $O(n)$ -bit numbers. Notice that the final output of a n -bit by n -bit multiply is $2n$ -bits wide.

There are other ways to perform multiplication by using repeated iterative steps. The naive approach is to have a single 1-bit multiplier, and on each cycle, generate an additional partial product. After n such steps, all of the partial products will have been generated. In parallel, the partial products can be added as they are generated with an accumulator. This uses considerably less hardware, but takes much longer to complete the calculation ($O(n)$).

In either iterative addition of partial products, or the usage of a Wallace Tree, the number of partial products is largely what determines how fast the multiplication can be performed. To reduce the number of partial products, the Booth algorithm can be used. This is a simple trick that involves the recoding the binary numbers using 0's, 1's and -1's. For example, the number 0011110_2 is the same as $01000\bar{1}0_2$, where $\bar{1}$ means -1. Any partial product that corresponds to a bit equalling zero can be skipped. This doesn't help much for the case where a tree of adders is used, but can save many iterations when an iterative method is used. To perform the encoding, start from the least significant bit. Each time a block of zero ends, and a block of ones starts, a $\bar{1}$ is written down. Each time a block of ones ends, and a block of zeros starts, a 1 is written down. If neither condition holds, a zero is written. The following is an example:

00011000011111 0 ← implicit zero
0010 $\bar{1}$ 00010000 $\bar{1}$

Implicitly, there is a zero to the right of the LSB. Blocks may be of size 1. In this example, 7 partial products are reduced to 4. In some cases the number of terms are reduced, but in other terms, there is no gain (consider alternating 0's and 1's).

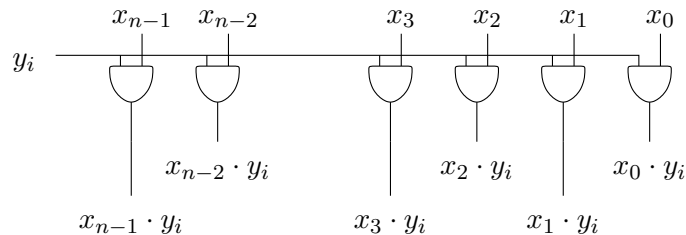


Figure 1: A n -bit by 1-bit multiply is achieved by using AND gates.

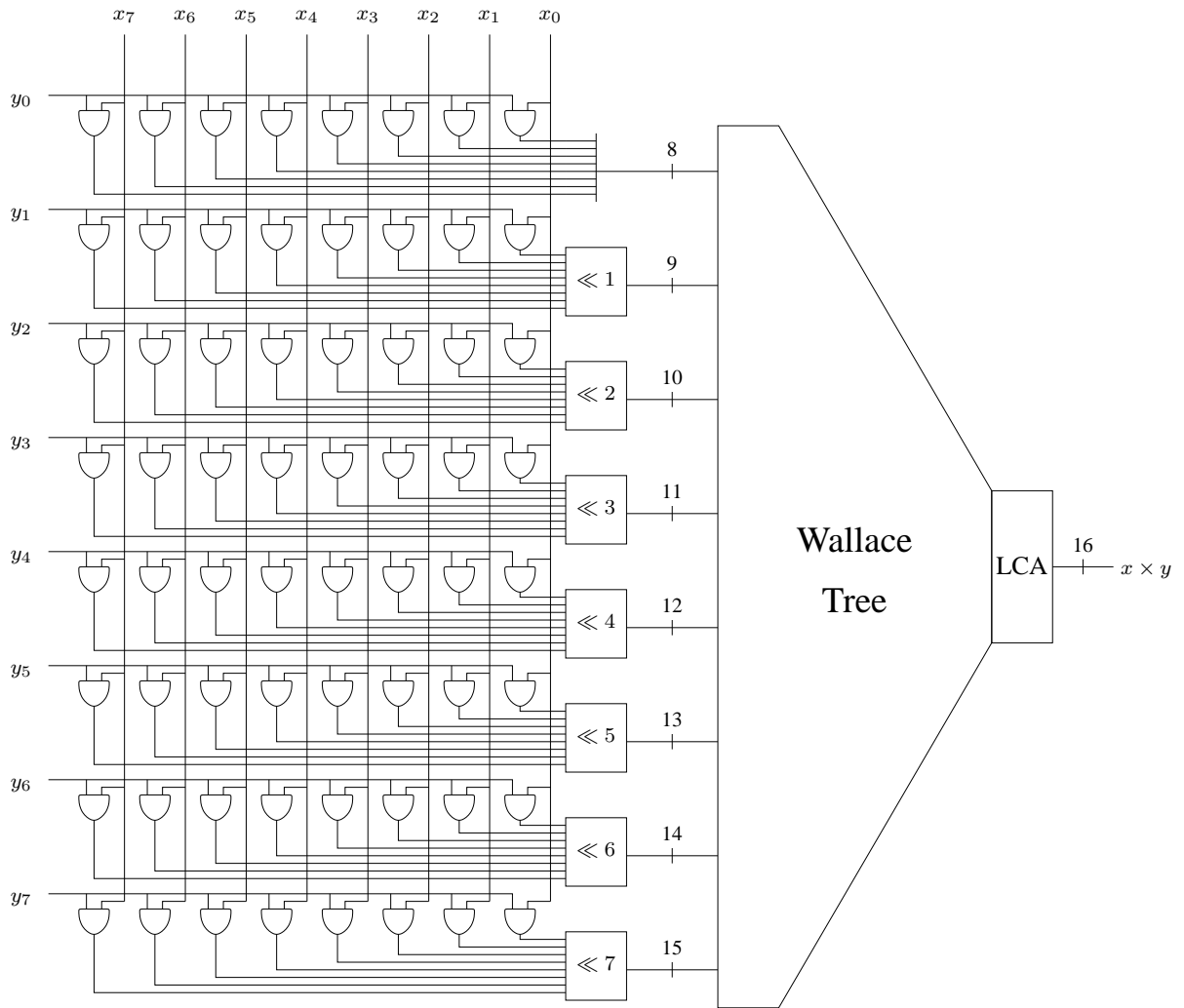


Figure 2: The n partial products can be computed by n separate 1-bit multiplies. The partial products are then combined with a Wallace Tree and LCA.

As before, we start with $\frac{R_0}{D}$. Then we repeatedly choose q_i such that $B(R_i - Dq_i) \in [q_{\perp}.q_{\perp}q_{\perp}q_{\perp}\dots, q_{\top}.q_{\top}q_{\top}q_{\top}\dots]$. In base 10 long division, $S = \{0, 1, \dots, 9\}$, and so the range limitation is simply that $10(R_i - Dq_i) \in [0.000\dots, 9.999\dots]$.

For all of our divisions, we are going to assume that both operands have been normalized such that the first bit is a one (i.e. $1.xxxx$). Renormalization can be performed at the end of the division to adjust the answer to be of the correct order of magnitude. At the very minimum, each step of the long division will require a comparison ($O(\lg n)$), a subtraction ($O(\lg n)$), and a shift ($O(1)$). With n iterations of this, we know that there will be a delay of at least $\Omega(n \lg n)$ with this approach. We have not discussed how the operation of choosing q_i to meet the specified requirements can be implemented efficiently either. In any case, we know that this approach for performing division takes longer than our target of $O(n)$ time.

2.2 SRT Division

SRT Division is named for the people who invented the algorithm. At about the same time, D. Sweeney (IBM), J.E. Robertson (University of Illinois) and T.D. Tocher (Imperial College of London) all independently discovered the algorithm. Before getting into the details, we will first look at the concept of *negative digits*. In base 10, we normally use the digits $S = \{0, 1, \dots, 9\}$ to represent numbers, and in this system, each number has a unique representation. Alternatively, we can use negative digits as well, such that $S = \{\bar{9}, \bar{8}, \dots, \bar{1}, 0, 1, \dots, 8, 9\}$, where $\bar{x} = -x$. Then, a number such as 16 can be represented as either 16 or $2\bar{4}$ ($20 + -4 = 16$). This is a redundant representation, which allows for more than one way to represent a number.

Radix 4 SRT Division

We use base 4 division (which allows us to process two bits per step), and we use the set of digits $S = \{\bar{2}, \bar{1}, 0, 1, 2\}$.

Algorithm:

```

 $R_0 := R$ 
for  $k = 0, 1, \dots$ 
    determine  $q_k \in S$  s.t.
         $R_{k+1} := 4(R_k - q_k D)$  and
         $|R_{k+1}| \leq \frac{8}{3}D$ 
end for

```

$$q = \frac{R}{D} = \sum_{i=0}^{\infty} \frac{q_i}{4^i}$$

This algorithm, as presented, should look pretty much like the abstract division algorithm presented earlier, except with the appropriate constants substituted in.

Theorem: The Radix 4 SRT Algorithm computes $q = \frac{R}{D} = \sum_{i=0}^{\infty} \frac{q_i}{4^i}$.

Proof:

$$R_{k+1} = 4(R_k - q_k D) \quad \text{by definition} \quad (1)$$

$$\frac{R_{k+1}}{D} = \frac{4R_k}{D} - 4q_k \quad \text{divide (1) by } D \quad (2)$$

$$\frac{4R_k}{D} = \frac{R_{k+1}}{D} + 4q_k \quad \text{rearrange terms} \quad (3)$$

$$\frac{R_k}{D} = \frac{R_{k+1}}{4D} + q_k \quad \text{divide by 4} \quad (4)$$

$$\frac{R}{D} = \frac{R_0}{D} \quad \text{by definition}$$

$$= \frac{R_1}{4D} + q_0 \quad \text{by (4)}$$

$$= \frac{1}{4} \left(\frac{R_2}{4D} + q_1 \right) + q_0 \quad \text{by (4)}$$

$$= \frac{1}{4^2} \left(\frac{R_2}{D} \right) + \frac{1}{4} q_1 + q_0 \quad \text{expansion of terms}$$

\vdots

$$= \frac{1}{4^k} \left(\frac{R_k}{D} \right) + \left(\frac{q_{k-1}}{4^{k-1}} + \dots + \frac{q_1}{4} + q_0 \right) \quad (5)$$

Since $-\frac{8}{3}D \leq R_k \leq \frac{8}{3}D$, then:

$$\lim_{k \rightarrow \infty} \frac{1}{4^k} \cdot \frac{R_k}{D} = \lim_{k \rightarrow \infty} \frac{1}{4^k} \cdot \frac{8}{3} = 0 \quad \text{substitute } R_k \quad (6)$$

$$\begin{aligned} \frac{R}{D} &= \lim_{k \rightarrow \infty} \frac{1}{4^k} \cdot \frac{R_k}{D} + \sum_{i=0}^{\infty} \frac{q_i}{4^i} \\ &= \sum_{i=0}^{\infty} \frac{q_i}{4^i} \quad \text{Q.E.D.} \quad (7) \end{aligned}$$

First question: why in the world do we use a limit of $\frac{8}{3}$?

Recall that in base 10, we required $R_{k+1} < q_{\top} \cdot q_{\top} \cdot q_{\top} \cdot q_{\top} \dots < 10$ (for $q_{\top} = 9$). In our radix 4 SRT division, $q_{\top} = 2$. So we have the condition that

$$\begin{aligned} R_{k+1} &< 2.2222\dots \quad (\text{base 4}) \\ &= 2 \left(\left(\frac{1}{4} \right)^0 + \left(\frac{1}{4} \right)^1 + \left(\frac{1}{4} \right)^2 + \dots \right) \\ &= \sum_{i=0}^{\infty} \left(\frac{1}{4} \right)^i \\ &= 2 \left(\frac{1}{1 - \frac{1}{4}} \right) \\ &= \frac{8}{3} \end{aligned}$$

Similarly for the lower bound of $-\frac{8}{3}$ using $q_{\perp} = \bar{2}$.

Second question: how do we choose the q_k 's?

Let's look at the different possible cases for the q_k 's.

$q = 2$: We start with the condition on the interval in which the remainder must always fall in, and then just perform some substitutions. The substituted value for q is underlined.

$$\begin{aligned} -\frac{8}{3}D &< R_{k+1} < \frac{8}{3}D \\ -\frac{8}{3}D &< 4(R_k - \underline{2} \cdot D) < \frac{8}{3}D \\ \frac{4}{3}D &< R_k < \frac{8}{3}D \end{aligned}$$

Therefore, if $R_k \in [\frac{4}{3}D, \frac{8}{3}D]$, we can choose $q_k = 2$.

$q = 1$:

$$\begin{aligned} -\frac{8}{3}D &< 4(R_k - \underline{1} \cdot D) < \frac{8}{3}D \\ \frac{1}{3}D &< R_k < \frac{5}{3}D \end{aligned}$$

Therefore, if $R_k \in [\frac{1}{3}D, \frac{5}{3}D]$, we can choose $q_k = 1$.

$q = 0$:

$$\begin{aligned} -\frac{8}{3}D &< 4(R_k - \underline{0} \cdot D) < \frac{8}{3}D \\ -\frac{2}{3}D &< R_k < \frac{2}{3}D \end{aligned}$$

Therefore, if $R_k \in [-\frac{2}{3}D, \frac{2}{3}D]$, we can choose $q_k = 0$.

$q = -1$:

$$\begin{aligned} -\frac{8}{3}D &< 4(R_k - \underline{-1} \cdot D) < \frac{8}{3}D \\ -\frac{5}{3}D &< R_k < -\frac{1}{3}D \end{aligned}$$

Therefore, if $R_k \in [-\frac{5}{3}D, -\frac{1}{3}D]$, we can choose $q_k = -1$.

$q = -2$:

$$\begin{aligned} -\frac{8}{3}D &< 4(R_k - \underline{-2} \cdot D) < \frac{8}{3}D \\ -\frac{8}{3}D &< R_k < -\frac{4}{3}D \end{aligned}$$

Therefore, if $R_k \in [-\frac{8}{3}D, -\frac{4}{3}D]$, we can choose $q_k = -2$.

Notice that there is some overlap in the ranges. For instance, if $R_k = \frac{1}{2}D$, then choosing either $q_k = 0$ or $q_k = 1$ will be correct. This may seem odd, but recall that when we use negative digits, there can be more than one way to represent a number. The intervals are plotted on the number line in Figure 3.

Third Question: now how do we *quickly* figure out which q_k we want?

The approach we'll use is to use a lookup table. Figure 4 shows a lookup table that takes a value for R and D , and returns an appropriate q_k . Notice that lines corresponding to the boundaries of the intervals shown in Figure 3 have been plotted in the table. Also, since we are normalizing the arguments, we don't actually have to include any of the

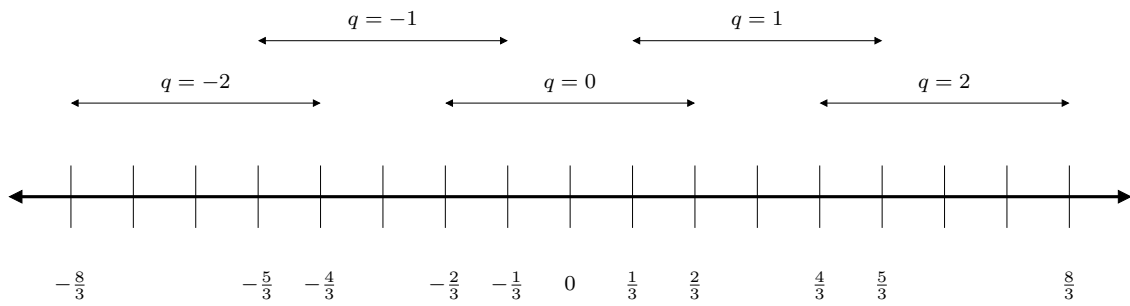


Figure 3: The redundant representation of numbers allows for more than one choice for the quotient digits.

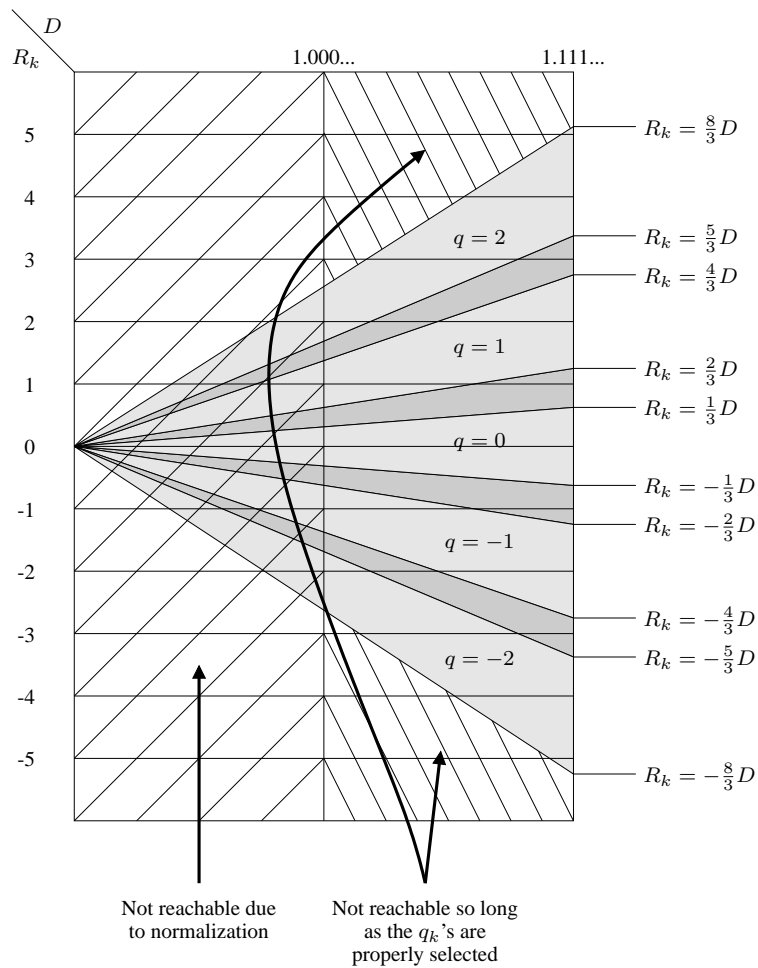


Figure 4: A lookup table for R_k and D can make choosing a suitable q_k only require $O(1)$ time. Certain regions of the table can never be reached.

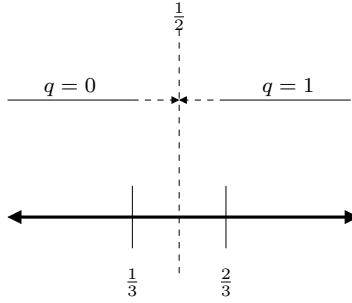


Figure 5: We will choose the midpoint in the overlap region to decide what value to return for q_k .

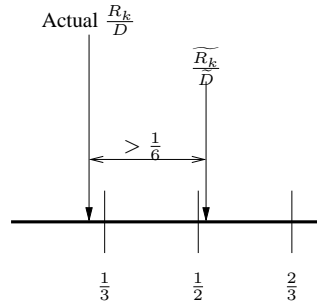


Figure 6: For a cutoff of $\frac{1}{2}$, an error greater than $\frac{1}{6}$ can cause erroneous quotient digits to be returned.

entries that are less than 1.000... or greater than 1.111.... With such a table, the choice for a particular q_k can be made in $O(1)$ time. The regions that allow for more than one choice for q_k are darker than the others.

An obvious problem with this approach is that this table will be of infinite size, and so it is not anywhere near realistic. The solution to this problem is to take advantage of the “slack” in what number we choose for q_k . In Figure 4, there are regions where we have a choice of what value for q_k we return. Let us choose the midpoint of the overlap region for deciding what to return (Figure 5). For example, for the $q = 0/q = 1$ overlap region, if $\frac{R_k}{D} > \frac{1}{2}$, then we’ll return $q_k = 1$, otherwise if $\frac{R_k}{D} \leq \frac{1}{2}$, we’ll return $q_k = 0$. Now, suppose we simply approximate the value of $\frac{R_k}{D}$ by $\frac{\tilde{R}_k}{\tilde{D}}$. If $\frac{\tilde{R}_k}{\tilde{D}}$ is less than $\frac{1}{6}$ away from $\frac{R_k}{D}$, we’ll still return the correct value of q_k . If the error is greater than $\frac{1}{6}$, erroneous values may be returned. An example is illustrated in Figure 6, where the actual value of $\frac{R_k}{D}$ dictates that $q_k = 0$ *must* be returned (i.e. it’s not in the overlap region), but the approximation has a sufficiently large error to result in $q_k = 1$ being returned. Our approximation for R_k and D is to simply use the first 8 bits of R_k and the first 5 bits of D . This results in an error that is less than $\frac{1}{6}$. At the same time, because \tilde{R}_k and \tilde{D} are of a constant size, we can now create a lookup table for all of the possible $\approx 2^5 \cdot 2^8$ inputs. In the Pentium, the cutoff in the overlap regions is not symmetrically located (i.e. not at the midpoint), which allows their implementation to only use 7 bits for \tilde{R}_k .

The encoding of $q \in S = \{2, \bar{1}, 0, 1, 2\}$ requires three bits. Instead, we will use two numbers to keep the positive and negative portions separate. In base 10, instead of $\bar{24}$, we’ll store 20 and $\bar{04}$ as two separate numbers. In the very end, these numbers can be combined by subtracting the negative portion from the positive part. The other trick that we will use is that all addition/subtraction through each iteration will be carried out in carry-save form, so we will never have to pay the price for a carry propagation. The only exception is when we need to compute \tilde{R}_k and \tilde{D} we will have to perform an addition to get the non-carry-save form of the numbers, but both of these have fixed widths that are independent of the number of bits in our arguments n , and so take $O(1)$ time.

Now let's go through an example in some detail. Let's divide 5506153 by 294911. In binary, 5506153 = 10101000000010001101001, and 294911 = 10001111111111111111. The first step is to normalize our arguments:

$$\begin{aligned} 10101000000010001101001 &= 1.0101000000010001101001 \times 2^{22} \\ 10001111111111111111 &= 1.0001111111111111110000 \times 2^{18} \end{aligned}$$

So the division that we will want to perform is:

$$\begin{aligned} R_0 &= \frac{1.0101000000010001101001}{1.0001111111111111110000} \times 2^{22-18} \\ D &= \frac{1.0101000000010001101001}{1.0001111111111111110000} \times 2^{22-18} \end{aligned}$$

The steps for each iteration are:

- A** determine q_k from the table
- B** calculate $-q_k D$
- C** add $R_k + -q_k D$
- D** multiply by 4 to get R_{k+1}

Iteration 1:

- A** $\tilde{R}_k = 0001.0101$, $\tilde{D} = 1.0001$, and so $q_0 = 1$
- B** compute $-q_k D = -D$. At this point we start using the carry-save format

$$\begin{aligned} -qD &= 1110.1110000000000000001111 && \text{invert bits, sign extend} \\ + &0000.0000000000000000000001 && \text{add 1, but in the carry} \end{aligned}$$

- C** add $R + -qD$

$$\begin{aligned} R &= 0001.0101000000010001101001 \\ -qD &= 1110.1110000000000000001111 \\ \text{partial sum} &= 1111.1011000000010001100110 \\ \text{carry} &= 0000.100000000000000001001\text{1} && \text{the 1 is from } -qD. \end{aligned}$$

- D** multiply by 4 (shift bits by 2)

$$\begin{aligned} \text{partial sum} &= 1111.1011000000010001100110 \\ \text{carry} &= 0000.1000000000000000010001 \\ &\Rightarrow \\ \text{new partial sum} &= 1110.1100000001000110011000 \\ \text{new carry} &= 0010.0000000000000000100010 \end{aligned}$$

Iteration 2:

- A** $\tilde{R}_k = 0000.1100$, \tilde{D} is the same, and so $q_1 = 1$. So far, we have $q_{\text{so far}} = 1.1$

B compute $-q_k D = -D$. Same as before, negate and add 1 in the carry.

C add $R + -qD$

$$\begin{aligned}
 \text{previous partial sum} &= 1110.1100000001000110011000 \\
 \text{previous carry} &= 0010.0000000000000001000100 \\
 -qD &= 1110.1110000000000000001111 \\
 \\
 \text{partial sum} &= 0010.0010000001000111010011 \\
 \text{carry} &= 1101.10000000000000001100\underline{1} \quad \text{the } \underline{1} \text{ is from } -qD.
 \end{aligned}$$

D multiply by 4 (shift bits by 2)

$$\begin{aligned}
 \text{partial sum} &= 0010.0010000001000111010011 \\
 \text{carry} &= 1101.100000000000000011001 \\
 &\Rightarrow \\
 \text{new partial sum} &= 1000.1000000100011101001100 \\
 \text{new carry} &= 0110.00000000000000001100100
 \end{aligned}$$

Iteration 3:

A $\tilde{R}_k = 1110.1000$, \tilde{D} is the same, and so $q_2 = \bar{1}$. So far, we have $q_{\text{so far}} = 1.1\bar{1}$

Recall that we stated that the digits of q would be stored as two separate numbers, one for the positive digits, and one for the negative digits. So $1.1\bar{1}$ is stored as

$$\begin{array}{rcccc}
 & & q_0 & q_1 & q_2 \\
 q_+ = \text{positive digits:} & 01. & 01 & 00 \\
 q_- = \text{negative digits:} & 00. & 00 & 01 \\
 \hline
 q = \text{difference:} & 01. & 00 & 11
 \end{array}$$

We will stop our example at this point since this becomes very tedious very quickly. Let us now analyze the cost for each iteration.

A Compute \tilde{R}_k and perform lookup: $O(1)$

B Compute $-q_k D$:

- $q = 2$: shift to get $2D$, invert and add 1 (in the carry) to get $-2D$: $O(1)$
- $q = 1$: invert and add 1 to get $-D$: $O(1)$
- $q = 0$: multiply by zero gives all zeros: $O(1)$
- $q = -1$: $-(-1)D = D$, do nothing to D : $O(1)$
- $q = -2$: $-(-2)D = 2D$, shift D : $O(1)$

C use carry save add to sum $R_k + -q_k D$: $O(1)$

D shift by 4: $O(1)$

So the total time per iteration is simply $O(1)$. For an n -bit answer, we'll have to run through the loop $\frac{n}{2} = O(n)$ times (recall that two bits are generated per iteration because we're using a radix 4 division). Now we must add up all of the other steps that occur before and after the main loop:

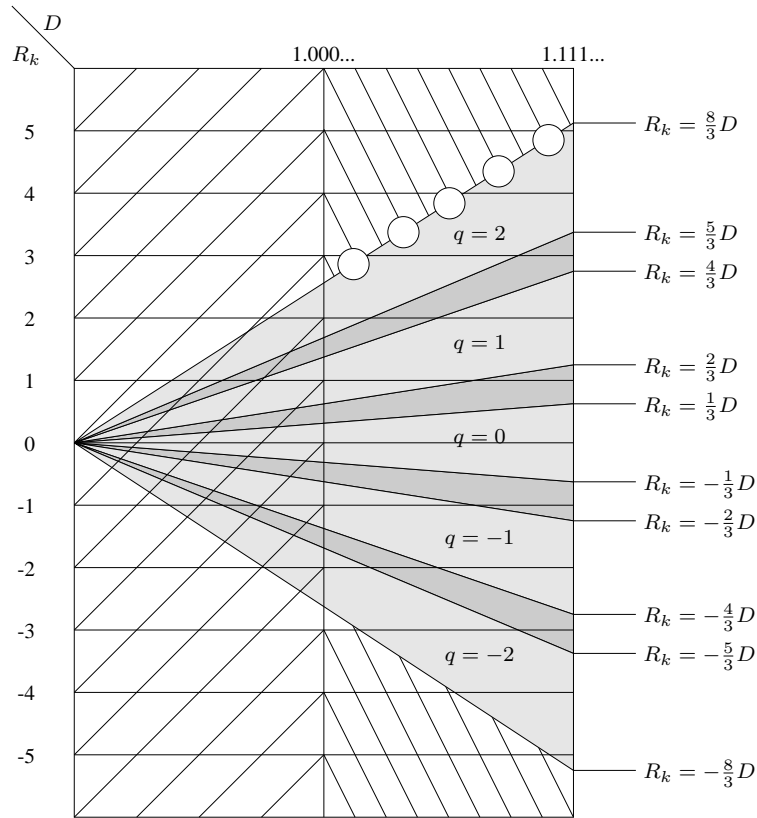


Figure 7: The Pentium's SRT lookup table contains five locations, marked by the \bigcirc 's, that would erroneously return 0 instead of 2.

- Normalization of arguments at the start: $O(\lg n)$
- $O(n)$ iterations of the loop at $O(1)$ per iteration: $O(n)$
- Final subtraction of $q_+ - q_-$ using LCA: $O(\lg n)$
- Renormalize: $O(\lg n)$

So the total time to perform an n -bit division is $O(n)$ when using the SRT algorithm.

2.3 The Pentium Division Bug

SRT division is used in Intel's Pentium processor (as well as most other processors that support division). The problem with the Pentium's implementation of SRT division is that the lookup table contains a few cells that would return incorrect values. The approximate location of the cells are illustrated in Figure 7. These are all located along the $\tilde{R}_k = \frac{8}{3}\tilde{D}$ line.

Because there are many cells in the table that can never be reached, no space is actually allocated for those entries. The table is simply hardwired to return a zero if any of those locations are ever accessed. Apparently, someone at Intel thought that five of the cells would never be accessed, and removed them, thus allowing some further optimizations of the table. It turns out that under very special circumstances, these cells can be accessed. At this point, the table should return $q_k = 2$, but a zero is returned instead.

Tim Coe (Vitesse Semiconductor Corporation) and Ping Tak Tang (Argonne National Laboratory) published a paper titled "It Takes Six Ones To Reach a Flaw". In the paper, they provide a proof that shows that the divisor must

contain six consecutive ones, starting from the fifth bit after the radix point. Another result from the paper is that an error cell can not be encountered before the ninth iteration, and therefore the first eight quotient digits will always be accurate. This explains why the Pentium bug did not really affect everyday computer users.

References:

- [1] Deley, David. 1995. "The Pentium Division Flaw". <http://home1.gte.net/deleyd/pentbug/pentbug.txt>
- [2] Edelman, Alan. 1997. "The Mathematics of the Pentium Bug". *SIAM Review* 39(March):54-67

The example used in this section was taken from [1].