Convergence Methods
Ivor Page[1]

14.1 Division by Convergence

In this section we consider methods generally used in floating point division algorithms that have *log* time complexity and that use multiply. Recall that the previous methods were based on shift and add/subtract. These asymptotically faster methods can also be used for integer division, but the very high speed of CSA addition and the use of radix-4 or higher radix methods makes the shift/add division algorithms competitive for integer divide. When large numbers of bits are required for the quotient, convergence methods become most suitable.

14.1.1 Newton Raphson Division

We wish to compute the reciprocal:

$$x = \frac{1}{B}$$

where B is a positive normalized binary fraction, $0.5 \leq B < 1$. Newton Raphsons' method is applied as follows:

$$\begin{aligned} f(x) &= \frac{1}{x} - B \\ x_{i+1} &= x_i - \frac{f(x_i)}{f'(x_i)} \\ &= x_i - \left[\frac{(1/x_i) - B}{-(1/x^2)}\right] \\ &= x_i(2 - Bx_i) \end{aligned}$$

Given an initial approximation for $x_0 = (\frac{1}{B} + \epsilon)$

$$\begin{aligned} x_1 &= (\frac{1}{B} + \epsilon)\left(2 - B(\frac{1}{B} + \epsilon)\right) \\ &= \frac{1}{B} - B\epsilon^2 \\ x_2 &= \frac{1}{B} - B^3\epsilon^4 \end{aligned}$$

A seed ROM of size 2^p entries is used to provide p bits of an initial approximation.

[1]University of Texas at Dallas

The method roughly doubles the number of valid quotient digits on each iteration. To get a reciprocal of k bits, $\lceil log_p\ k \rceil$ iterations are required. Here is the setup for the table:

$$B = 0.1b_2b_3 \cdots b_p \cdots b_k$$

The p bit address presented to the ROM is $b_2b_3 \cdots b_{p+1}$. The p bit ROM output is $d_1d_2 \cdots d_p$ and the initial approximation is $x_0 = 1.d_1d_2 \cdots d_p$.

There is one more simplifying step that leads to an efficient algorithm:

Let $a_i = Bx_i$, then:

$$\begin{aligned} x_{i+1} &= x_i \times (2 - a_i) \\ a_{i+1} &= a_i \times (2 - a_i) \end{aligned}$$

The common factor $(2 - a_1)$ simplifies the algorithm and the hardware. Initially $a_0 = Bx_0$.

The algorithm requires a 2's complement operation and two multiplies in each iteration, except the last, in which only one multiply is needed. For a normalized binary fraction a_i, $(2 - a_i)$ is simply the 2's complement of a_i.

$$\lim_{i\to\infty} x_i = \frac{1}{B} \qquad \lim_{i\to\infty} a_i = 1.0$$

Here is an example assuming a 4 bit ROM table:

```
   B = 0.110101011101  =  0.835205
  x0 = 1.0011              Initial 4 bit approximation
  a0 =  0.1111110111100111
2-a0 = 1.0000001000011001
  x1 = 1.0011
  a1 = 0.1111111111111011
2-a1 = 1.0000000000000100
  x2 = 1.00110010
  a2 = 0.1111111111111111
2-a2 = 1.0000000000000000
  x3 = 1.001100101000
  a3 = 1.
2-a3 = 1.
  x4 = 1.0011001010000010 = 1.19731
```

Since we begin with only p bits for x_0 from the table, in the first iteration the multiply operations can be constrained to $p \times p$ bits. In the second iteration, $2p \times 2p$ bit multiplies

will be needed, and so on, until the required number of bits is reached. This observation enables a speedup since the shift-add multiply algorithms have $O(k)$ running time. If a redundant number system, or a tree multiplier is used, the running time of the multiply operation is $O(log\ k)$. In either case, a speedup is possible if the size of the intermediate results is constrained in this way. Obtaining this speedup in practice requires fairly complex additions to most of the hardware schemes that we have studied for multiplication.

14.1.2 Round-off Errors in Newton Raphson Divide

Such errors are introduced by a fixed multiplier width. If we multiply two k bit binary fractions (or integers) and round the result to k bits, the result is in error by up to 1/2 *ulp*. If we truncate the result, the error is up to 1 *ulp*. These errors can accumulate from iteration to iteration. Consider a simple analysis in which an error ϵ is added on every multiplication. Notice in the example above that a_i rapidly approaches 1.0. Once $a_i \approx 1.0$, then

$$x_{i+1} \quad = \quad x_i + \epsilon$$

Up to 1 *ulp* is added on each iteration. A simple programming example will show this to be true. Let's repeat the example above assuming that the result of each multiply is truncated to 9 binary places.

```
Input B: 0.110101011101

  B = 0.11010101110100000 : 0.835205
1/B = 1.00110010100000101 : 1.19731
 x0 = 1.0011

i |       x, no truncation        | x with truncation
--|-------------------------------|----------------------
1 | 1.00110010011111011 : 1.19723 | 1.001100100 : 1.19531
2 | 1.00110010100000101 : 1.19731 | 1.001100101 : 1.19727
3 | 1.00110010100000101 : 1.19731 | 1.001100110 : 1.19922
4 | 1.00110010100000101 : 1.19731 | 1.001100111 : 1.20117
5 | 1.00110010100000101 : 1.19731 | 1.001101000 : 1.20313
6 | 1.00110010100000101 : 1.19731 | 1.001101001 : 1.20508
7 | 1.00110010100000101 : 1.19731 | 1.001101010 : 1.20703
8 | 1.00110010100000101 : 1.19731 | 1.001101011 : 1.20898
9 | 1.00110010100000101 : 1.19731 | 1.001101100 : 1.21094
```

In the right-hand columns, where truncation to 9 binary places takes place after each multiply, an additive error of 1 in the 9th fractional digit is seen on each iteration.

If we want a result accurate to k bits, then all arithmetic should be done using $k + log_2\ m$ bits, where m iterations are needed. To see this, assume that 1 *ulp* is added on each iteration and 4 iterations are needed. Then $log_2\ 4 = 2$ extra bits would be used. The contents of these two extra bits would progress through the sequence $00, 01, 10, 11$ before the error carried into the lsb of the k bit result.

14.1.3 Multiplicative Division

We begin with the original problem and expand it by multiplication of the dividend and divisor by a series of functions, x_i:

$$\begin{aligned} q &= \frac{z}{d} \\ &= \frac{z \times x_1 \times x_2 \cdots \times x_m}{d \times x_1 \times x_2 \cdots \times x_m} \\ \text{such that} & \\ & d \times x_1 \times x_2 \cdots \times x_m \to 1.0 \\ & z \times x_1 \times x_2 \cdots \times x_m \to q \\ \text{as} \quad & m \to \infty \end{aligned}$$

Initially the divisor is normalized such that $0.5 \leq d < 1.0$ and the dividend is adjusted to match.

Convergence depends on the selection of the sequence of values x_i.

Call the i^{th} iteration of the dividend $d_i = d \times x_1 \times \cdots \times x_i$. Initially $d = 1 + \delta$. Successive multipliers, x_i, are chosen such that $d_{i+1} > d_i$. We choose $x_1 = 1 - \delta = 2 - d$. Then $d_1 = 1 - \delta^2$. Then choose $x_2 = 1 + \delta^2$ such that $d_2 = 1 - \delta^4$. In general, $x_i = 1 + \delta^{2^{(i-1)}}, \ \ i > 1$.

Consider the product sequence

$$\begin{aligned} & d \times x_1 \times x_2 \times x_3 \times \cdots \\ = \ & d \times (1 - \delta) \times (1 + \delta^2) \times (1 + \delta^4) \times \cdots \\ = \ & d \times (1 - \delta + \delta^2 - \delta^3) \times (1 + \delta^4) \times \cdots \\ S \ = \ & d(1 - \delta + \delta^2 - \delta^3 + \delta^4 - \delta^5 + \delta^6 - \cdots) \\ S\delta \ = \ & d(\delta - \delta^2 + \delta^3 - \delta^4 + \delta^5 - \delta^6 - \cdots) \\ S + S\delta \ = \ & d \\ S \ = \ & \frac{d}{1 + \delta} \\ S \ = \ & 1 \end{aligned}$$

Here is the algorithm:

$$
\begin{array}{l}
q = z \\
while(d < 0.111\cdots1)\{ \\
\quad x = 2 - d \\
\quad d = d \times x \\
\quad q = q \times x \\
\}
\end{array}
$$

As with Newton Raphson, there is a 2's complement operation and two multiplies in each iteration.

Here is an example, given in decimals for simplicity:

```
                          d0= 0.6215          q0= 0.7795
 i=1   x =  1.378500000   d = 0.8567377500    q = 1.074540750
 i=2   x =  1.143262250   d = 0.9794759277    q = 1.228481876
 i=3   x =  1.020524072   d = 0.9995787625    q = 1.253695326
 i=4   x =  1.000421238   d = 0.9999998226    q = 1.254223430
 i=5   x =  1.000000177   d = 1.0000000000    q = 1.254223652
```

Here is the same problem with values in binary:

```
                       d0= 0.1001111100011010 q0= 0.1100011110001101
i=1 x = 1.011000001110 d = 0.1101101101010011 q = 1.0001001100010101
i=2 x = 1.001001001010 d = 0.1111101010111110 q = 1.0011101001111101
i=3 x = 1.000001010100 d = 0.1111111111100100 q = 1.0100000011110010
i=4 x = 1.000000000001 d = 0.1111111111111111 q = 1.0100000100010100
i=5 x = 1.000000000000 d = 0.1111111111111111 q = 1.0100000100010100
```

The algorithm converges most slowly when $d \approx 1/2$. Then $(1-d_i) = 1/2, 1/4, 1/16, 1/256, 1/65536, \cdots$. The quotient is accurate to $1, 2, 4, 8, 16, \cdots$ bits. The time complexity is $log_2 k$, larger than for Newton Raphson by a constant factor. A table lookup to get the algorithm started would provide p bits of accuracy and would result in similar running time to that of Newton Raphson. Both algorithms are said to have *quadratic convergence.*

14.1.4 Round-off Error in Division by Repeated Multiplication

In each iteration, we perform $x = 2 - d$, followed by $d = d \times x$. An additive error of 1 *ulp* in the multiplication gets carried into the next iteration and leads to a final error of m *ulps* overall, where m iterations are performed.

As with Newton Raphson, the final error can be bounded by 1 *ulp* by doing all the calculations in $k + log_2\ m$ bits.

14.2 Square Root Extraction by Convergence

The Newton Raphson methods can also be applied to the problem of square root extraction:

$$\begin{aligned} f(x) &= x^2 - a = 0 \\ x_{i+1} &= x_i - \frac{f(x)}{f'(x)} \\ &= x_i - \frac{x_i^2 - a}{2x_i} \\ &= \frac{1}{2}(x_i + \frac{a}{2x_i}) \end{aligned}$$

This scheme would require a division in each iteration. Alternatively we can compute the reciprocal of the radicand, $x = 1/\sqrt{a}$ and multiply the result by a:

$$\begin{aligned} f(x) &= \frac{1}{x^2} - a = 0 \\ x_{i+1} &= x_i - \frac{f(x)}{f'(x)} \\ &= x_i + \frac{1}{2}(x_i - ax_i^3) \\ &= \frac{1}{2}x_i(3 - ax_i^2) \end{aligned}$$

$$\text{then } \sqrt{a} = a \times x_m$$

Here is an example:

```
Input a = 1.345,
 x0 = 2/(1+a) = 0.852878 = 0.1101101001

 i:1   x =  0.8621086337 = 0.1101110010110011
 i:2   x =  0.8622611866 = 0.1101110010111101
 i:3   x =  0.8622612271 = 0.1101110010111101

    Root = a*x = 1.15974135
```

The method requires 3 multiplies and one addition in each iteration. Quadratic convergence provides a 64 bit result in only 2 or 3 iterations if an initial approximation of 8 to 10 bits is

found by table lookup. The initial approximation computed above is accurate to 6.07% or less. This translates into about 4 bits of accuracy.

The lookup table should contain values of $1.5x_0$ and $0.5x_0$ to save computations in the first iteration.