Fast division: concluding the unfinished symphony of computer arithmetic

Guy Even

SEE-NERGIA talk
Division of Floating Point Numbers

Number represented by sign $s \in \{0, 1\}$, exponent $e \in \mathbb{Z}$, and fraction $f \in [1, 2)$:

$$(-1)^s \cdot 2^e \cdot f$$
Division of Floating Point Numbers

- Number represented by sign $s \in \{0, 1\}$, exponent $e \in \mathbb{Z}$, and fraction $f \in [1, 2)$:
  
  $$(-1)^s \cdot 2^e \cdot f$$

- Division of two floating point numbers:

  $$\frac{(-1)^{s_1} \cdot 2^{e_1} \cdot f_1}{(-1)^{s_2} \cdot 2^{e_2} \cdot f_2} = (-1)^{s_1-s_2} \cdot 2^{e_1-e_2} \cdot \frac{f_1}{f_2}$$
Division of Floating Point Numbers

Number represented by sign \( s \in \{0, 1\} \), exponent \( e \in \mathbb{Z} \), and fraction \( f \in [1, 2) \):

\[
(-1)^s \cdot 2^e \cdot f
\]

Division of two floating point numbers:

\[
\frac{(-1)^{s_1} \cdot 2^{e_1} \cdot f_1}{(-1)^{s_2} \cdot 2^{e_2} \cdot f_2} = (-1)^{s_1-s_2} \cdot 2^{e_1-e_2} \cdot \frac{f_1}{f_2}
\]

Main difficulty in computing \( f_1/f_2 \).
Division

“School Method”: long division requires 1 subtraction per bit, so delay is at least $\Omega(n \log n)$. 

Improve by using redundant representation so that subtraction requires constant delay (allows “wrong” guesses). Constant time per bit division with $O(n)$ delay.

Other improvements (increase radix) still require linear time. Used in many microprocessors! Faster algorithms can be obtained based on Newton iterations.
“School Method” : long division requires 1 subtraction per bit, so delay is at least $\Omega(n \log n)$.

Improve by using redundant representation so that subtraction requires constant delay (allows “wrong” guesses). Constant time per bit $\Rightarrow$ division with $O(n)$ delay.
Division

- “School Method”: long division requires 1 subtraction per bit, so delay is at least $\Omega(n \log n)$.

- Improve by using redundant representation so that subtraction requires constant delay (allows “wrong” guesses). Constant time per bit $\Rightarrow$ division with $O(n)$ delay.

- Other improvements (increase radix) still require linear time. Used in many microprocessors!
Division

“School Method” : long division requires 1 subtraction per bit, so delay is at least $\Omega(n \log n)$.

Improve by using redundant representation so that subtraction requires constant delay (allows “wrong” guesses). Constant time per bit $\Rightarrow$ division with $O(n)$ delay.

Other improvements (increase radix) still require linear time. Used in many microprocessors!

Faster algorithms can be obtained based on Newton iterations.
<table>
<thead>
<tr>
<th>processor</th>
<th>ALU</th>
<th>FP add</th>
<th>FP mult</th>
<th>FP div single</th>
<th>FP div double</th>
</tr>
</thead>
<tbody>
<tr>
<td>ULTRA-Sparc 3</td>
<td>1</td>
<td>4(1)</td>
<td>4(1)</td>
<td>17(15)</td>
<td>20(18)</td>
</tr>
<tr>
<td>Pentium 3</td>
<td>1</td>
<td>3(1)</td>
<td>5(2)</td>
<td>17(17)</td>
<td>32(32)</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>1</td>
<td>5(1)</td>
<td>7(2)</td>
<td>23(23)</td>
<td>38(38)</td>
</tr>
<tr>
<td>Itanium</td>
<td>1</td>
<td>5(1)</td>
<td>5(1)</td>
<td>30+(11)*</td>
<td>40+(13)*</td>
</tr>
<tr>
<td>AMD Athlon</td>
<td>1</td>
<td>4(1)</td>
<td>4(1)</td>
<td>16(13)</td>
<td>20(17)</td>
</tr>
<tr>
<td>Power3</td>
<td>1</td>
<td>4(1)</td>
<td>4(1)</td>
<td>17(13)</td>
<td>21(17)</td>
</tr>
<tr>
<td>Motorola G4</td>
<td>1</td>
<td>5(1)</td>
<td>5(1)</td>
<td>21(21)</td>
<td>35(35)</td>
</tr>
<tr>
<td>Alpha 21064</td>
<td>1</td>
<td>4(1)</td>
<td>4(1)</td>
<td>34(34)</td>
<td>63(63)</td>
</tr>
<tr>
<td>Alpha 21164</td>
<td>1</td>
<td>4(1)</td>
<td>4(1)</td>
<td>19(19)</td>
<td>31(31)</td>
</tr>
<tr>
<td>Alpha 21264/21364</td>
<td>1</td>
<td>4(1)</td>
<td>4(1)</td>
<td>12(9)</td>
<td>15(12)</td>
</tr>
<tr>
<td>R8000</td>
<td>1</td>
<td>4(1)</td>
<td>4(1)</td>
<td>14(11)</td>
<td>20(17)</td>
</tr>
<tr>
<td>R12000</td>
<td>1</td>
<td>2(1)</td>
<td>2(1)</td>
<td>14(12)</td>
<td>21(19)</td>
</tr>
<tr>
<td>Proposed Divider (1/2)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>9(6)</td>
<td>11(8)</td>
</tr>
<tr>
<td>Proposed Divider (1)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>9(3)</td>
<td>11(4)</td>
</tr>
<tr>
<td>Proposed Divider (2)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>9(1)</td>
<td>11(2)</td>
</tr>
<tr>
<td>Proposed Divider (3)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>9(1)</td>
<td>11(1)</td>
</tr>
</tbody>
</table>
\( A/B \) with Newton iterations

- idea: (1) reciprocal: \( x = \frac{1}{B} \) (2) multiply: \( Q = A \cdot x \).
A/B with Newton iterations

- idea: (1) reciprocal: $x = \frac{1}{B}$ (2) multiply: $Q = A \cdot x$.
- Reciprocal computation using Newton iterations for

$$f(x) = B - \frac{1}{x}.$$ 

Root of $f(x) = 0$ is $\frac{1}{B}$. 


\[ A/B \text{ with Newton iterations} \]

- idea: (1) reciprocal: \( x = \frac{1}{B} \) (2) multiply: \( Q = A \cdot x \).
- Reciprocal computation using Newton iterations for

\[ f(x) = B - \frac{1}{x}. \]

Root of \( f(x) = 0 \) is \( \frac{1}{B} \).

- Newton iterations: an initial estimate \( x_0 \neq 0 \) and iterate

\[
x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}
= x_i - \frac{B - 1/x_i}{1/x_i^2}
= x_i - B \cdot x_i^2 + x_i
= x_i \cdot (2 - B \cdot x_i).
\]
Reciprocal Computation $1/B$ with Newton iterations

Initial estimate $x_0$, and $x_{i+1} = x_i \cdot (2 - B \cdot x_i)$.
Reciprocal Computation $1/B$ with Newton iterations

Initial estimate $x_0$, and $x_{i+1} = x_i \cdot (2 - B \cdot x_i)$. 

$f(x) = \frac{1}{x} - B$
Error analysis: Newton iterations

- Initial estimate $x_0$, and $x_{i+1} = x_i \cdot (2 - B \cdot x_i)$. 
Error analysis : Newton iterations

- Initial estimate $x_0$, and $x_{i+1} = x_i \cdot (2 - B \cdot x_i)$.
- Consider the relative error term $e_i$ defined by

$$e_i \triangleq \frac{1}{B} - x_i = 1 - B \cdot x_i.$$
Error analysis: Newton iterations

- Initial estimate $x_0$, and $x_{i+1} = x_i \cdot (2 - B \cdot x_i)$.
- Consider the relative error term $e_i$ defined by

$$e_i \triangleq \frac{1}{B} - \frac{x_i}{1/B} = 1 - B \cdot x_i.$$

- It follows that

$$e_{i+1} = 1 - B \cdot x_{i+1}$$

$$= 1 - B \cdot x_i \cdot (2 - B \cdot x_i)$$

$$= (1 - B \cdot x_i)^2 = e_i^2.$$
Error analysis (cont)

- Initial estimate $x_0$, and

\[
x_{i+1} = x_i \cdot (2 - B \cdot x_i)
\]
\[
e_{i+1} = e_i^2.
\]
Error analysis (cont)

- Initial estimate $x_0$, and

\[ x_{i+1} = x_i \cdot (2 - B \cdot x_i) \]

\[ e_{i+1} = e_i^2. \]

- Implications:

1. If initial error $e_0 < 1$, then
   \[ x_i \neq 1 \cdot B. \]
2. Quadratic convergence rate: number of accurate bits doubles in every iteration
   \[ \text{after } \log n \text{ iterations we have } n \text{ bits of the reciprocal.} \]
3. \[ e_{i+1} = 0 \] implies \[ x_{i+1} = 1 \cdot B \] (one sided convergence).
Error analysis (cont)

- Initial estimate $x_0$, and

$$x_{i+1} = x_i \cdot (2 - B \cdot x_i)$$

$$e_{i+1} = e_i^2.$$

- Implications:
  1. If initial error $e_0 < 1$, then $x_i \to \frac{1}{B}$. 

\[ - p.8 \]
Error analysis (cont)

- Initial estimate $x_0$, and

$$x_{i+1} = x_i \cdot (2 - B \cdot x_i)$$

$$e_{i+1} = e_i^2.$$

- Implications:
  1. If initial error $e_0 < 1$, then $x_i \to \frac{1}{B}$.
  2. Quadratic convergence rate: number of accurate bits doubles in every iteration $\Rightarrow$ after $\log n$ iterations we have $n$ bits of the reciprocal.
Error analysis (cont)

- Initial estimate $x_0$, and

\[ x_{i+1} = x_i \cdot (2 - B \cdot x_i) \]
\[ e_{i+1} = e_i^2. \]

- Implications:
  1. If initial error $e_0 < 1$, then $x_i \to \frac{1}{B}$.
  2. Quadratic convergence rate: number of accurate bits doubles in every iteration $\Rightarrow$ after $\log n$ iterations we have $n$ bits of the reciprocal.
  3. $e_{i+1} \geq 0$ implies $x_{i+1} \leq \frac{1}{B}$ (one sided convergence).
Error analysis (cont-2)

- Initial estimate $x_0$, and

\[
x_{i+1} = x_i \cdot (2 - B \cdot x_i)
\]

\[
e_{i+1} = e_i^2.
\]
Error analysis (cont-2)

- Initial estimate $x_0$, and

$$x_{i+1} = x_i \cdot (2 - B \cdot x_i)$$

$$e_{i+1} = e_i^2.$$

- Numerical stability: what happens if intermediate computations are not precise?
Error analysis (cont-2)

- Initial estimate $x_0$, and

$$x_{i+1} = x_i \cdot (2 - B \cdot x_i)$$
$$e_{i+1} = e_i^2.$$

- Numerical stability: what happens if intermediate computations are not precise?
  - $D_i := B \cdot x_i + \varepsilon_1$
Error analysis (cont-2)

- Initial estimate $x_0$, and

\[ x_{i+1} = x_i \cdot (2 - B \cdot x_i) \]

\[ e_{i+1} = e_i^2. \]

- Numerical stability: what happens if intermediate computations are not precise?

\[ D_i := B \cdot x_i + \varepsilon_1 \]

\[ F_i := 2 - D_i + \varepsilon_2 \]
Error analysis (cont-2)

- Initial estimate $x_0$, and

\[
x_{i+1} = x_i \cdot (2 - B \cdot x_i)
\]
\[
e_{i+1} = e_i^2.
\]

- Numerical stability: what happens if intermediate computations are not precise?

- $D_i := B \cdot x_i + \varepsilon_1$
- $F_i := 2 - D_i + \varepsilon_2$
- $x_{i+1} := x_i \cdot F_i + \varepsilon_3$

Not hard to analyze error since algorithm "recovers" from errors!
Error analysis (cont-2)

- Initial estimate $x_0$, and

\[
x_{i+1} = x_i \cdot (2 - B \cdot x_i)
\]

\[
e_{i+1} = e_i^2.
\]

- Numerical stability: what happens if intermediate computations are not precise?

- $D_i := B \cdot x_i + \varepsilon_1$
- $F_i := 2 - D_i + \varepsilon_2$
- $x_{i+1} := x_i \cdot F_i + \varepsilon_3$

- Not hard to analyze error since algorithm “recovers” from errors!
Pipelining: Newton iterations

Each step requires 3 operations:

- \( D_i := B \cdot x_i \)
- \( F_i := 2 - D_i \)
- \( x_{i+1} := x_i \cdot F_i \)
Pipelining: Newton iterations

Each step requires 3 operations:

- \( D_i := B \cdot x_i \)
- \( F_i := 2 - D_i \)
- \( x_{i+1} := x_i \cdot F_i \)

2 dependent multiplications per iteration... slows down computation.
Recap - division with Newton

- \( \log n \) iterations compute \( 1/B \).
- Each iteration requires 2 dependent multiplications.
- \( \Rightarrow O(\log^2 n) \) delay for computing \( 1/B \).
- Final multiplication \( A \cdot (1/B) \) gives quotient.
Recap - division with Newton

- $\log n$ iterations compute $1/B$.
- Each iteration requires 2 dependent multiplications.
- $\Rightarrow O(\log^2 n)$ delay for computing $1/B$.
- final multiplication $A \cdot (1/B)$ gives quotient.

Q: parallelize/pipeline multiplications in each iteration?
Enabling parallelization

Newton iterations

\[(*) \quad x_{i+1} = x_i \cdot F_i,\]

where

\[D_i \triangleq B \cdot x_i\]

\[F_i \triangleq 2 - D_i.\]
Enabling parallelization

Newton iterations

\[(*) \quad x_{i+1} = x_i \cdot F_i,\]

where

\[D_i \triangleq B \cdot x_i\]

\[F_i \triangleq 2 - D_i.\]

Goldschmidt’s algorithm [1964]
Enabling parallelization

Newton iterations

\((*)\) \[ x_{i+1} = x_i \cdot F_i, \]

where

\[ D_i \triangleq B \cdot x_i \]

\[ F_i \triangleq 2 - D_i. \]

Goldschmidt’s algorithm [1964]

Define

\[ N_i \triangleq A \cdot x_i. \]
Enabling parallelization

Newton iterations

\[(*) \quad x_{i+1} = x_i \cdot F_i,\]

where

\[D_i \triangleq B \cdot x_i\]

\[F_i \triangleq 2 - D_i.\]

Goldschmidt’s algorithm [1964]

Define

\[N_i \triangleq A \cdot x_i.\]

Multiply both sides of \((*)\) by \(A\) \& \(B\):

\[
\begin{aligned}
A \cdot x_{i+1} &= A \cdot x_i \cdot F_i \\
B \cdot x_{i+1} &= B \cdot x_i \cdot F_i
\end{aligned}
\]
Enabling parallelization

Newton iterations

\[ x_{i+1} = x_i \cdot F_i, \]

where

\[ D_i \triangleq B \cdot x_i \]
\[ F_i \triangleq 2 - D_i. \]

Goldschmidt’s algorithm [1964]

Define

\[ N_i \triangleq A \cdot x_i. \]

Multiply both sides of (*) by \( A \) & \( B \):

\[
\begin{align*}
A \cdot x_{i+1} &= A \cdot x_i \cdot F_i \\
B \cdot x_{i+1} &= B \cdot x_i \cdot F_i
\end{align*}
\]

\[ \Leftrightarrow \]

\[
\begin{align*}
N_{i+1} &= N_i \cdot F_i \\
D_{i+1} &= D_i \cdot F_i
\end{align*}
\]
Goldschmidt - properties

Since

\[ N_i \triangleq A \cdot x_i \]
\[ D_i \triangleq B \cdot x_i \]
\[ x_i \to 1/B, \]

it follows that

\[ N_i \to A/B \]
\[ D_i \to 1. \]

Convergence rate - same as Newton iterations! (only if intermediate computations are precise)
Goldschmidt’s algorithm - listing

Require: $|e_0| < 1$.

1: Initialize:

\[
N_{-1} := A \\
D_{-1} := B \\
F_{-1} := \frac{1 - e_0}{B}.
\]

2: for $i = 0$ to $k$ do
3: \( N_i := N_{i-1} \cdot F_{i-1} \).
4: \( D_i := D_{i-1} \cdot F_{i-1} \).
5: \( F_i := 2 - D_i \).
6: end for
7: Return($N_i$)
Parallelization

![Diagram of parallelization process with iterative equations and algorithm steps]

- **Approximation (1/B)**:
  - Iteration 0: $e_0 \leq \hat{e}_0$
  - Iteration 1: $e_1 \leq \hat{e}_0^2$
  - Iteration $k-1$: $e_{k-1} \leq \hat{e}_0^{2k-1}$
  - Iteration $k$: $e_{k-1} \leq \hat{e}_0^{2k}$

- **D, F-Pipeline**:
  - Iteration 0:
    - $D_{-1} := B$
    - $F_{-1} := \text{APPROX}(1/B)$
  - Iteration 0:
    - $N_{-1} := A$

- **N-Pipeline**:
  - Iteration 0:
    - $N_{0} := N_{-1} \cdot F_{-1}$
    - $F_{0} := 2 - D_{0}$
  - Iteration 0:
    - $D_{0} := D_{-1} \cdot F_{-1}$

- **Algorithm Steps**:
  - **Initialization**:
    - $D_0 := X_0 \cdot B$
    - $F_0 := 2 - D_0$
  - **Iteration**:
    - $D_k := D_{k-1} \cdot F_{k-2}$
    - $F_k := 2 - D_k$
  - **Finalization**:
    - $N_k := X_k \cdot A$
Error analysis: Goldschmidt’s algorithm

Numerical stability: what happens if intermediate computations are not precise?
Error analysis: Goldschmidt’s algorithm

- Numerical stability: what happens if intermediate computations are not precise?
  - $D_{i+1} := D_i \cdot F_i + \varepsilon_1$
Error analysis: Goldschmidt’s algorithm

- Numerical stability: what happens if intermediate computations are not precise?

- $D_{i+1} := D_i \cdot F_i + \varepsilon_1$
- $N_{i+1} := N_i \cdot F_i + \varepsilon_2$
Error analysis: Goldschmidt’s algorithm

Numerical stability: what happens if intermediate computations are not precise?

- \( D_{i+1} := D_i \cdot F_i + \varepsilon_1 \)
- \( N_{i+1} := N_i \cdot F_i + \varepsilon_2 \)
- \( F_i := 2 - D_i + \varepsilon_2 \)
Error analysis (cont)

Newton iterations:

\[ (*) \quad x_{i+1} = x_i \cdot F_i, \]

multiplied by \( A \) & \( B \):

\[
\begin{align*}
A \cdot x_{i+1} &= A \cdot x_i \cdot F_i \\
B \cdot x_{i+1} &= B \cdot x_i \cdot F_i
\end{align*}
\]

Goldschmidt’s algorithm:

\[
\begin{align*}
N_{i+1} &= N_i \cdot F_i \\
D_{i+1} &= D_i \cdot F_i
\end{align*}
\]
Newton iterations:

\[ (*) \quad x_{i+1} = x_i \cdot F_i, \]

multiplied by \( A \) & \( B \):

\[
\begin{align*}
A \cdot x_{i+1} &= A \cdot x_i \cdot F_i \\
B \cdot x_{i+1} &= B \cdot x_i \cdot F_i
\end{align*}
\]

Goldschmidt’s algorithm:

\[
\begin{align*}
N_{i+1} &= N_i \cdot F_i \\
D_{i+1} &= D_i \cdot F_i
\end{align*}
\]

Convergence based on invariant:

\[
\frac{N_i}{D_i} = \frac{A}{B}.
\]

Imprecise computations violate invariant, and

\[ N_i \not\to A/B. \]
Conclusion:

- Goldschmidt’s alg is not self-correcting.
Conclusion:

- Goldschmidt’s alg is not self-correcting.
- Bounding error used to be complicated:
Error analysis (cont 2)

Conclusion:

- Goldschmidt’s alg is not self-correcting.
- Bounding error used to be complicated:
  - IBM 360 model 91 [1967]: ad hoc error analysis.
Conclusion:

- Goldschmidt’s alg is not self-correcting.
- Bounding error used to be complicated:
  - IBM 360 model 91 [1967]: ad hoc error analysis.
  - AMD K7 [1999]: combining formal proof methods that span thousands of lines with millions of test vectors.
Conclusion:

- Goldschmidt’s alg is not self-correcting.
- Bounding error used to be complicated:
  - IBM 360 model 91 [1967]: ad hoc error analysis.
  - AMD K7 [1999]: combining formal proof methods that span thousands of lines with millions of test vectors.
- Pessimistic bounds imply larger multipliers that waste area, power, and increased delay.
Our contribution [E+Seidel+Ferguson]

A parametric error analysis of Goldschmidt’s algorithm.

- Allows different error bounds for every intermediate computation (so a sequence of increasing multipliers can be analyzed).

- Enables searching for optimal hardware tradeoffs (i.e., initial approximation and multiplier sizes in each stage).

- We showed that the analysis used in AMD-K7 is not tight - could use smaller multipliers and save 10% in overall cost of FP-DIV micro-architecture.

- Greatly simplifies the task of verification.
Our contribution [E+Seidel+Ferguson]

A parametric error analysis of Goldschmidt’s algorithm.

- Allows different error bounds for every intermediate computation (so a sequence of increasing multipliers can analyzed).

- Enables searching for optimal hardware tradeoffs (i.e., initial approximation & multiplier sizes in each stage).

We showed that analysis used in AMD-K7 is not tight - could use smaller multipliers and save 10% in overall cost of FP-DIV micro-architecture.

Greatly simplify task of verification.
Our contribution [E+Seidel+Ferguson]

A parametric error analysis of Goldschmidt’s algorithm.

- Allows different error bounds for every intermediate computation (so a sequence of increasing multipliers can analyzed).
- Enables searching for optimal hardware tradeoffs (i.e., initial approximation & multiplier sizes in each stage).
- We showed that analysis used in AMD-K7 is not tight - could use smaller multipliers and save 10% in overall cost of FP-DIV micro-architecture.
Our contribution [E+Seidel+Ferguson]

A parametric error analysis of Goldschmidt’s algorithm.
- Allows different error bounds for every intermediate computation (so a sequence of increasing multipliers can analyzed).
- Enables searching for optimal hardware tradeoffs (i.e., initial approximation & multiplier sizes in each stage).
- We showed that analysis used in AMD-K7 is not tight - could use smaller multipliers and save 10% in overall cost of FP-DIV micro-architecture.
- Greatly simplify task of verification.
More contributions [E+Seidel]

A complete description of an FP-DIV micro-architecture for single & double precision.

- Uses a half sized multiplier \((n \times (n/2))\) vs. \(n \times n\) both for double & single precision.
More contributions [E+Seidel]

A complete description of an FP-DIV micro-architecture for single & double precision.

- Uses a half sized multiplier \((n \times (n/2))\) vs. \(n \times n\) both for double & single precision.
- Smaller multiplier \(\Rightarrow\) shorter clock period, less area, less power.
More contributions [E+Seidel]

A complete description of an FP-DIV micro-architecture for single & double precision.

- Uses a half sized multiplier \( n \times (n/2) \) vs. \( n \times n \) both for double & single precision.

- Smaller multiplier \( \Rightarrow \) shorter clock period, less area, less power.

- Fewer cycles!