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 $\Omega(n \log n)$.

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.

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.

processor	latency				
	ALU	FP add	FP mult	FP div single	FP div double
ULTRA-Sparc 3	1	4(1)	4(1)	17(15)	20(18)
Pentium 3	1	3(1)	5(2)	17(17)	32(32)
Pentium 4	1	5(1)	7(2)	23(23)	38(38)
Itanium	1	5(1)	5(1)	30+(11)*	40+(13)*
AMD Athlon	1	4(1)	4(1)	16(13)	20(17)
Power3	1	4(1)	4(1)	17(13)	21(17)
Motorola G4	1	5(1)	5(1)	21(21)	35(35)
Alpha 21064	1	4(1)	4(1)	34(34)	63(63)
Alpha 21164	1	4(1)	4(1)	19(19)	31(31)
Alpha 21264/21364	1	4(1)	4(1)	12(9)	15(12)
R8000	1	4(1)	4(1)	14(11)	20(17)
R12000	1	2(1)	2(1)	14(12)	21(19)
Proposed Divider (1/2)	-	-	-	9(6)	11(8)
Proposed Divider (1)	-	-	-	9(3)	11(4)
Proposed Divider (2)	-	-	-	9(1)	11(2)
Proposed Divider (3)	-	-	-	9(1)	11(1)

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 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

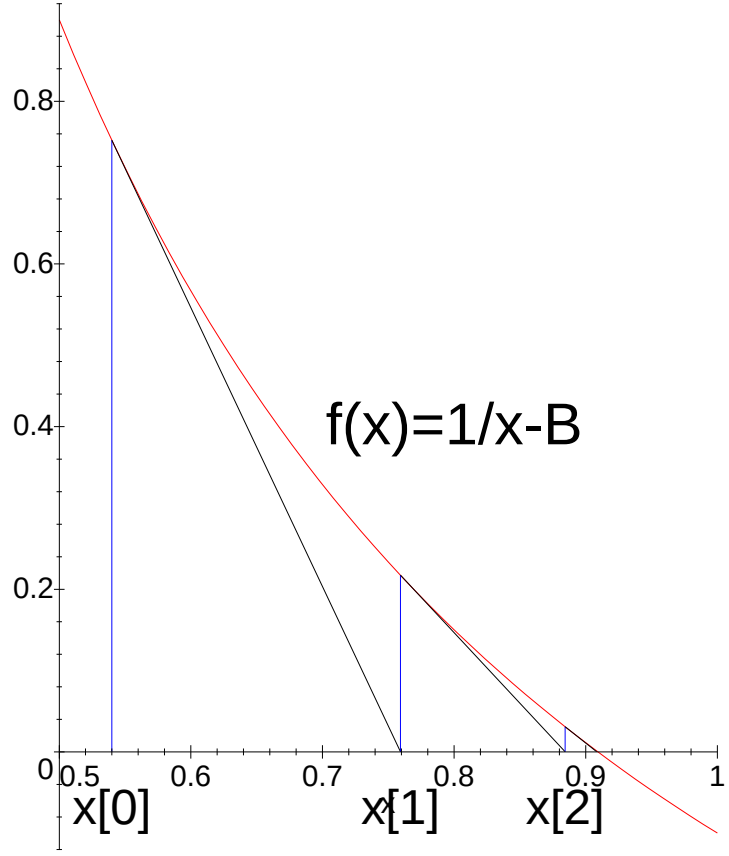
$$\begin{aligned} x_{i+1} &= x_i - \frac{f(x_i)}{f'(x_i)} \\ &= x_i - \frac{B - 1/x_i}{1/x_i^2} \\ &= x_i - B \cdot x_i^2 + x_i \\ &= x_i \cdot (2 - B \cdot x_i). \end{aligned}$$

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



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

$$e_i \stackrel{\triangle}{=} \frac{\frac{1}{B} - x_i}{\frac{1}{B}} = 1 - B \cdot x_i.$$

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

$$e_i \stackrel{\triangle}{=} \frac{\frac{1}{B} - x_i}{\frac{1}{B}} = 1 - B \cdot x_i.$$

- It follows that

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

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:

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 $x_i \rightarrow \frac{1}{B}$.

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

Error analysis (cont-2)

- Initial estimate x_0, and

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

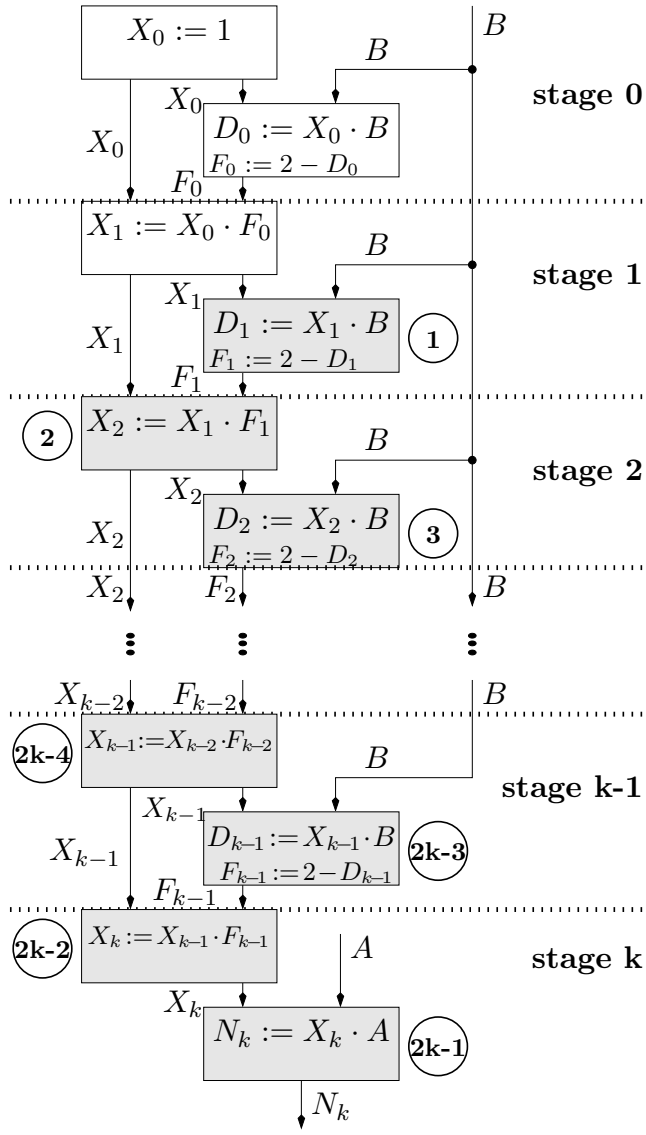
$$e_{i+1} = e_i^2.$$

- 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:

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

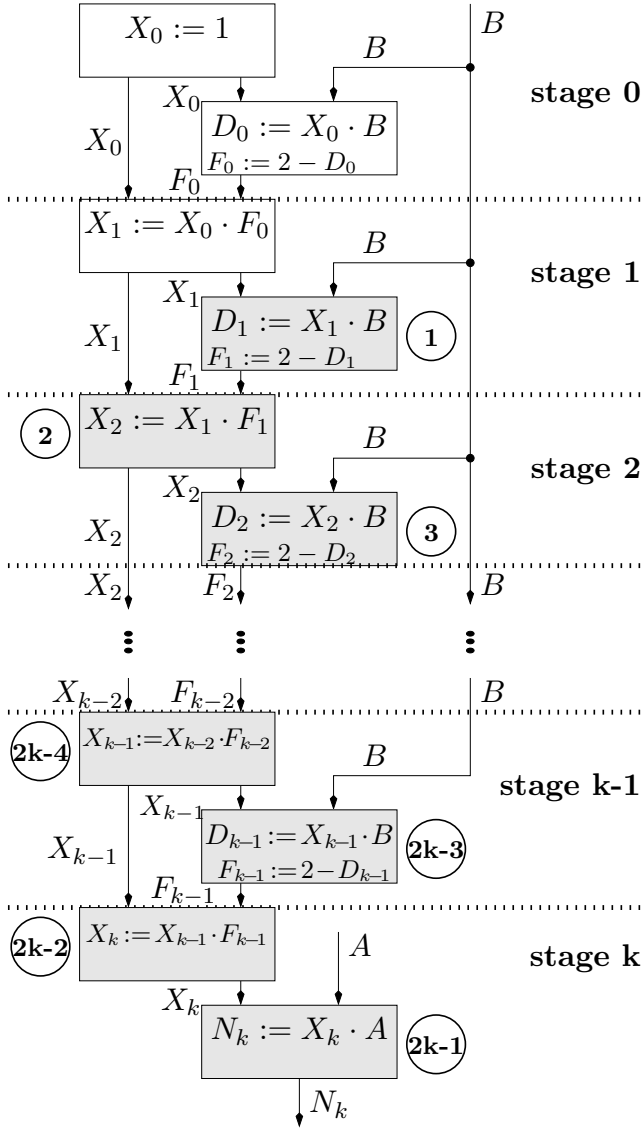


Pipelining : Newton iterations

each step requires 3 operations:

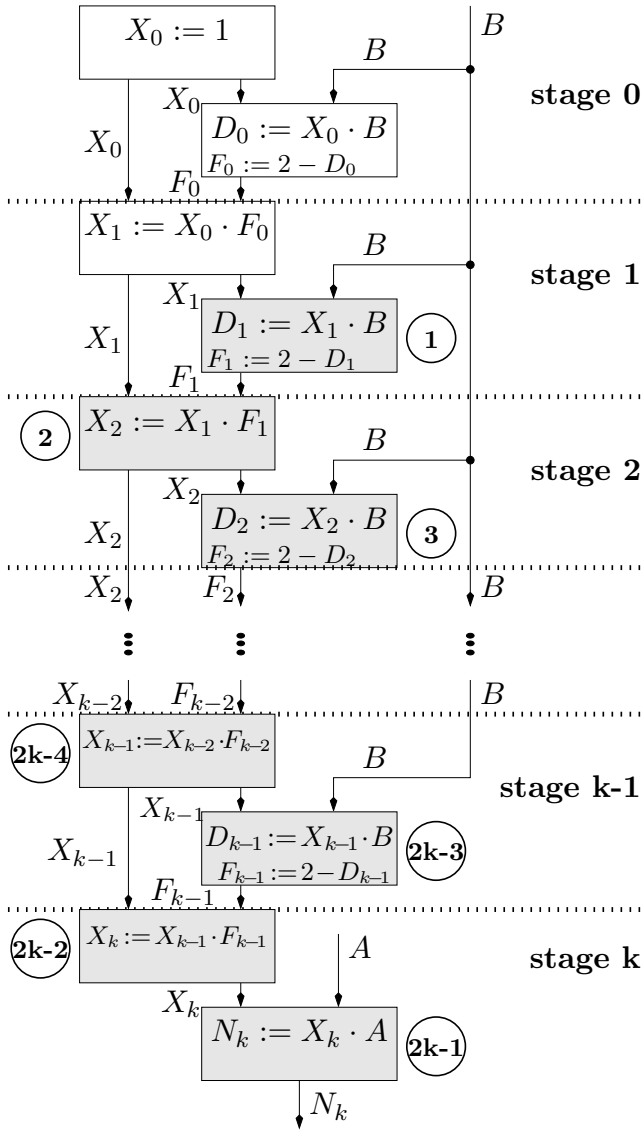
- $D_i := B \cdot x_i$
- $F_i := 2 - D_i$
- $x_{i+1} := x_i \cdot F_i$

2 dependent multiplications per iteration... slows down computation.



Recap - division with Newton

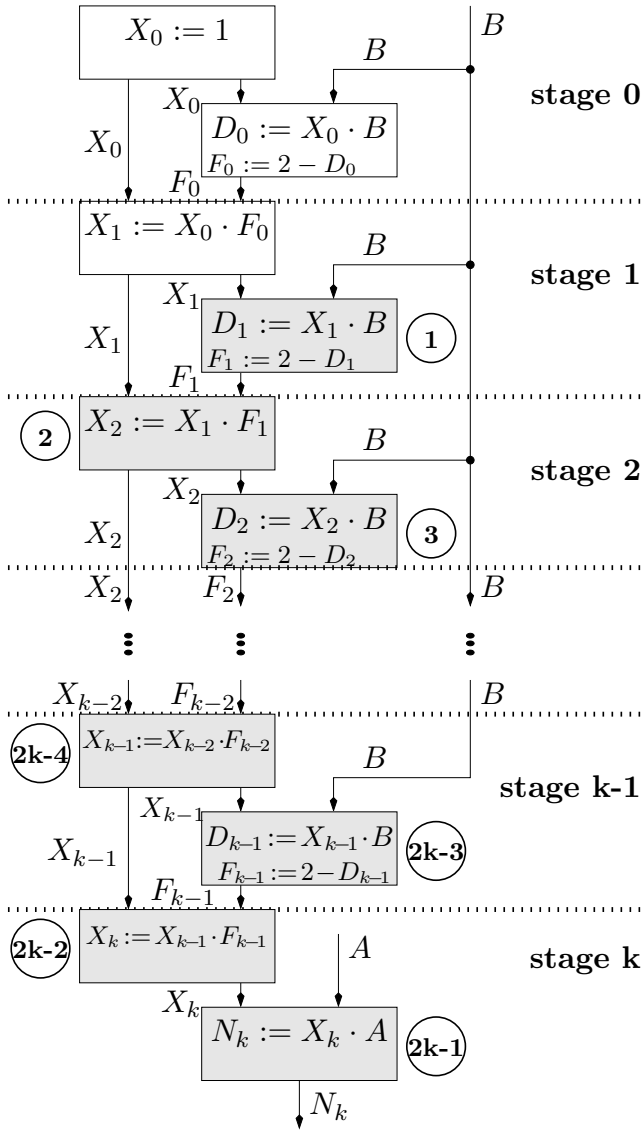
- $\log n$ iterations compute $1/B$.
- Each iteration requires 2 *dependent* multiplications.
- $\Rightarrow O(\log^2 n)$ delay for computing $1/B$.
- final multiplication $A \cdot (1/B)$ gives quotient.



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

$$D_i \stackrel{\triangle}{=} B \cdot x_i$$

$$F_i \stackrel{\triangle}{=} 2 - D_i.$$

Enabling parallelization

Newton iterations

Goldschmidt's algorithm [1964]

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

where

$$D_i \stackrel{\triangle}{=} B \cdot x_i$$

$$F_i \stackrel{\triangle}{=} 2 - D_i.$$

Enabling parallelization

Newton iterations

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

where

$$D_i \stackrel{\triangle}{=} B \cdot x_i$$

$$F_i \stackrel{\triangle}{=} 2 - D_i.$$

Goldschmidt's algorithm [1964]

Define

$$N_i \stackrel{\triangle}{=} A \cdot x_i.$$

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:

$$\begin{cases} A \cdot x_{i+1} = A \cdot x_i \cdot F_i \\ B \cdot x_{i+1} = B \cdot x_i \cdot F_i \end{cases}$$

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:

$$\begin{cases} A \cdot x_{i+1} = A \cdot x_i \cdot F_i \\ B \cdot x_{i+1} = B \cdot x_i \cdot F_i \end{cases}$$

$$\Leftrightarrow$$

$$\begin{cases} N_{i+1} = N_i \cdot F_i \\ D_{i+1} = D_i \cdot F_i \end{cases}$$

Goldschmidt - properties

Since

$$N_i \stackrel{\triangle}{=} A \cdot x_i$$

$$D_i \stackrel{\triangle}{=} B \cdot x_i$$

$$x_i \rightarrow 1/B,$$

it follows that

$$N_i \rightarrow A/B$$

$$D_i \rightarrow 1.$$

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: $\quad N_i := N_{i-1} \cdot F_{i-1}$.

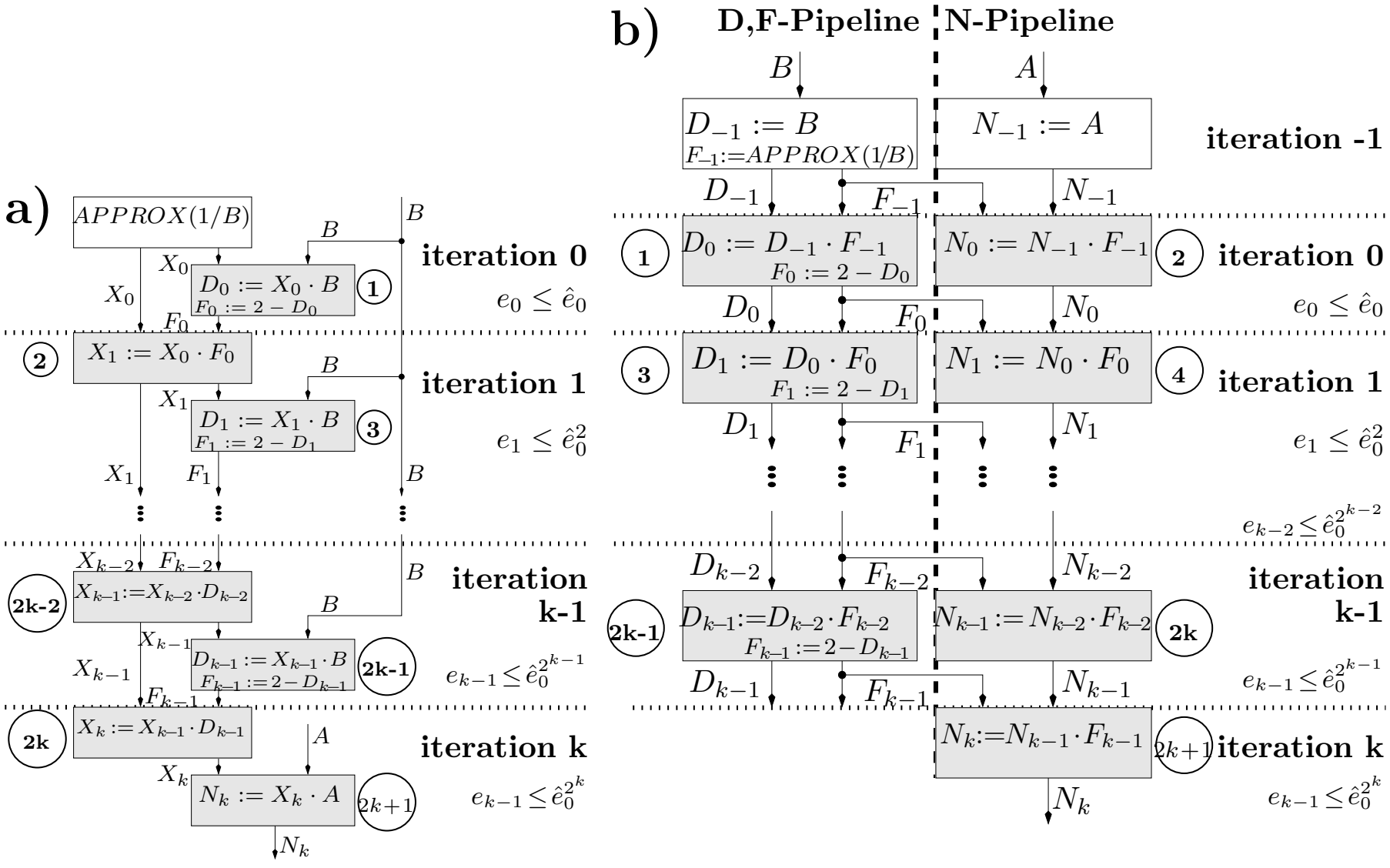
4: $\quad D_i := D_{i-1} \cdot F_{i-1}$.

5: $\quad F_i := 2 - D_i$.

6: **end for**

7: Return(N_i)

Parallelization



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:

$$\begin{cases} A \cdot x_{i+1} = A \cdot x_i \cdot F_i \\ B \cdot x_{i+1} = B \cdot x_i \cdot F_i \end{cases}$$

Goldschmidt's algorithm:

$$\begin{cases} N_{i+1} = N_i \cdot F_i \\ D_{i+1} = D_i \cdot F_i \end{cases}$$

Error analysis (cont)

Newton iterations:

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

multiplied by A & B:

$$\begin{cases} A \cdot x_{i+1} = A \cdot x_i \cdot F_i \\ B \cdot x_{i+1} = B \cdot x_i \cdot F_i \end{cases}$$

Goldschmidt's algorithm:

$$\begin{cases} N_{i+1} = N_i \cdot F_i \\ D_{i+1} = D_i \cdot F_i \end{cases}$$

Convergence based on invariant

$$\frac{N_i}{D_i} = \frac{A}{B}.$$

Imprecise computations violate invariant, and

$$N_i \not\to A/B.$$

Error analysis (cont 2)

Conclusion:

- Goldschmidt's alg is not self-correcting.

Error analysis (cont 2)

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.

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.
  - AMD K7 [1999]: combining formal proof methods that span thousands of lines with millions of test vectors.

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.
  - 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 analyzed).

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).

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!