Implementation of binary floating-point arithmetic on embedded integer processors

Polynomial evaluation-based algorithms and certified code generation

Guillaume Revy

ParLab EECS University of California, Berkeley

Ph.D. thesis’ work done under the direction of Claude-Pierre Jeannerod and Gilles Villard Arénaire INRIA project-team (LIP, Ens Lyon, France)
Motivation

- **Embedded systems** are ubiquitous
  - microprocessors dedicated to one or a few specific tasks
  - satisfy constraints: area, energy consumption, conception cost
Motivation

- **Embedded systems** are ubiquitous
  - microprocessors dedicated to one or a few specific tasks
  - satisfy constraints: area, energy consumption, conception cost

- Some embedded systems **do not have any FPU** (floating-point unit)
Motivation

- **Embedded systems** are ubiquitous
  - microprocessors dedicated to one or a few specific tasks
  - satisfy constraints: area, energy consumption, conception cost

- Some embedded systems do not have any FPU (floating-point unit)

- Highly used in audio and video applications
  - demanding on floating-point computations
Motivation

- **Embedded systems** are ubiquitous
  - microprocessors dedicated to one or a few specific tasks
  - satisfy constraints: area, energy consumption, conception cost

- Some embedded systems **do not have any FPU** (floating-point unit)

- Highly used in audio and video applications
  - demanding on floating-point computations
Motivation

- **Embedded systems** are ubiquitous
  - microprocessors dedicated to one or a few specific tasks
  - satisfy constraints: area, energy consumption, conception cost

- Some embedded systems **do not have any FPU** (floating-point unit)

- Highly used in audio and video applications
  - demanding on **floating-point computations**

---

- **ST231**
  - 4-issue VLIW 32-bit integer processor
  - At most 2 pipelined multiplications / cycle
  - Latencies: ALU = 1 cycle / Mul = 3 cycles

- Software implementing floating-point arithmetic
How to emulate floating-point arithmetic in software?

Design and implementation of efficient software support for IEEE 754 floating-point arithmetic on integer processors

- Existing software for IEEE 754 floating-point arithmetic:
  - Software floating-point support of GCC, Glibc and μClibc, GoFast Floating-Point Library
  - SoftFloat (→ STlib)
  - FLIP (Floating-point Library for Integer Processors)
    - software support for binary32 floating-point arithmetic on integer processors
    - correctly-rounded addition, subtraction, multiplication, division, square root, reciprocal, ...
    - handling subnormals, and handling special inputs
Towards the generation of fast and certified codes

- **Underlying problem:** development “by hand”
  - long and tedious, error prone
  - new target? new floating-point format?
Towards the generation of fast and certified codes

- **Underlying problem:** development “by hand”
  - long and tedious, error prone
  - new target? new floating-point format?
  ⇒ need for **automation and certification**
Towards the generation of fast and certified codes

**Underlying problem:** development “by hand”
- long and tedious, error prone
- new target? new floating-point format?
  ⇒ need for automation and certification

**Current challenge:** tools and methodologies for the automatic generation of efficient and certified programs
- optimized for a given format, for the target architecture
Towards the generation of fast and certified codes

- **Arénaire’s developments**: hardware (FloPoCo) and software (Sollya, Metalibm)

- **Spiral project**: hardware and software code generation for DSP algorithms

  *Can we teach computers to write fast libraries?*
Towards the generation of fast and certified codes

- Arénarie’s developments: hardware (FloPoCo) and software (Sollya, Metalibm)

- Spiral project: hardware and software code generation for DSP algorithms

  Can we teach computers to write fast libraries?

- Our tool: CGPE (Code Generation for Polynomial Evaluation)

  In the particular case of polynomial evaluation, we can teach computers to write fast and certified codes, for a given target and optimized for a given format.
Basic blocks for implementing correctly-rounded operators

(X, Y)

Special input detection

Floating-point number unpacking

Normalization

Range reduction

Result sign/exponent computation

Result significand approximation

Rounding condition decision

Correct rounding computation

Result reconstruction

Special output selection

function independent

function dependent

Objectives

→ Low latency, correctly-rounded implementations

→ ILP exposure
Basic blocks for implementing correctly-rounded operators

(X, Y)

Special input detection

Floating-point number unpacking

Normalization

Range reduction

Result sign/exponent computation

Result significand approximation

Rounding condition decision

Correct rounding computation

Result reconstruction

Special output selection

function independent

function dependent

Objectives

→ Low latency, correctly-rounded implementations

→ ILP exposure
Basic blocks for implementing correctly-rounded operators

- Floating-point number unpacking
- Normalization
- Range reduction
- Result sign/exponent computation
- Result significand approximation
- Rounding condition decision
- Correct rounding computation
- Result reconstruction
- Special input detection

■ Uniform approach for \( n \)th roots and their reciprocals
  → polynomial evaluation

■ Extension to division

\[ (X, Y) \]

\[ R \]
Flowchart for generating efficient and certified C codes

Problem: function to be evaluated

ST231 features

Computation of polynomial approximant

Efficient and certified C code generation

C code

Certificate
Flowchart for generating efficient and certified C codes

1. Problem: function to be evaluated
2. Computation of polynomial approximant
3. Efficient and certified C code generation
4. ST231 features

Constraints:
- Accuracy of approximant and C code
Flowchart for generating efficient and certified C codes

Problem: function to be evaluated

Computation of polynomial approximant

Efficient and certified C code generation

ST231 features

Constraints

- Accuracy of approximant and C code
- Low evaluation latency on ST231, ILP exposure
Flowchart for generating efficient and certified C codes

Problem: function to be evaluated

Computation of polynomial approximant

Efficient and certified C code generation

C code
Certificate

ST231 features

Constraints

- Accuracy of approximant and C code
  - Sollya

- Low evaluation latency on ST231, ILP exposure
Flowchart for generating efficient and certified C codes

- Problem: function to be evaluated
- Computation of polynomial approximant
- Efficient and certified C code generation
- C code
- Certificate
- ST231 features

Constraints
- Accuracy of approximant and C code
  - Sollya
  - interval arithmetic (MPFI), Gappa
- Low evaluation latency on ST231, ILP exposure
Flowchart for generating efficient and certified C codes

- Problem: function to be evaluated
- Efficient and certified C code generation
- Computation of polynomial approximant
- Efficient and certified C code
generation
- ST231 features
- Constraints
  - Accuracy of approximant and C code
    - Sollya
    - interval arithmetic (MPFI), Gappa
  - Low evaluation latency on ST231, ILP exposure
    - ?
Flowchart for generating efficient and certified C codes

Problem: function to be evaluated

Efficient and certified C code generation

Computation of polynomial approximant

ST231 features

Constraints

- Accuracy of approximant and C code
  - Sollya
  - interval arithmetic (MPFI), Gappa
- Low evaluation latency on ST231, ILP exposure
  - ?
- Efficiency of the generation process
Outline of the talk

1. Design and implementation of floating-point operators
   - Bivariate polynomial evaluation-based approach
   - Implementation of correct rounding

2. Low latency parenthesization computation
   - Classical evaluation methods
   - Computation of all parenthesizations
   - Towards low evaluation latency

3. Selection of effective evaluation parenthesizations
   - General framework
   - Automatic certification of generated C codes

4. Numerical results

5. Conclusions

6. And now in ParLab: Debugging of floating-point programs
Outline of the talk

1. Design and implementation of floating-point operators
   - Bivariate polynomial evaluation-based approach
   - Implementation of correct rounding

2. Low latency parenthesization computation

3. Selection of effective evaluation parenthesizations

4. Numerical results

5. Conclusions

6. And now in ParLab: Debugging of floating-point programs
Notation and assumptions

- Input \((x, y)\) and output \(\text{RN}(x/y)\): normal numbers
  - no underflow nor overflow
  - precision \(p\), extremal exponents \(e_{\text{min}}, e_{\text{max}}\)

\[
x = \pm 1.m_{x,1} \ldots m_{x,p-1} \cdot 2^{e_x} \quad \text{with} \quad e_x \in \{e_{\text{min}}, \ldots, e_{\text{max}}\}
\]
Notation and assumptions

- Input \((x, y)\) and output \(\text{RN}(x/y)\): normal numbers
  - no underflow nor overflow
  - precision \(p\), extremal exponents \(e_{\text{min}}, e_{\text{max}}\)
  - \(x = \pm 1.m_{x,1} \ldots m_{x,p-1} \cdot 2^{e_x} \quad \text{with} \quad e_x \in \{e_{\text{min}}, \ldots, e_{\text{max}}\}\)
  - RoundTiesToEven
Notation and assumptions

- **Standard binary encoding**: $k$-bit unsigned integer $X$ encodes input $x$

  - $s_x$: 1 bit
  - $E_x = e_x - e_{\text{min}} - 1$: $w = k - p$ bits
  - $T_x = m_{x,1} \ldots m_{x,p-1}$: \( p - 1 \) bits

- **Computation**: $k$-bit unsigned integers

  $\rightarrow$ integer and fixed-point arithmetic

\[ (X, Y) \xrightarrow{\text{Division C code}} R \]
Notation and assumptions

- **Standard binary encoding**: $k$-bit unsigned integer $X$ encodes input $x$

  \[
  s_x \quad E_x = e_x - e_{\text{min}} - 1 \quad T_x = m_{x,1} \ldots m_{x,p-1}
  \]

  \[
  \begin{align*}
  s_x & \quad 1 \text{ bit} \\
  E_x & \quad w = k - p \text{ bits} \\
  T_x & \quad p - 1 \text{ bits}
  \end{align*}
  \]

- **Computation**: $k$-bit unsigned integers

  $\rightarrow$ integer and fixed-point arithmetic
Range reduction of division

Express the exact result $r = x/y$ as:

$$ r = \ell \cdot 2^d \Rightarrow \text{RN}(x/y) = \text{RN}(\ell) \cdot 2^d $$

with

$$ \ell \in [1, 2) \quad \text{and} \quad d \in \{e_{\text{min}}, \ldots, e_{\text{max}}\} $$
Range reduction of division

- Express the exact result \( r = x/y \) as:
  \[
  r = \ell \cdot 2^d \quad \Rightarrow \quad \text{RN}(x/y) = \text{RN}(\ell) \cdot 2^d
  \]
  with
  \[
  \ell \in [1, 2) \quad \text{and} \quad d \in \{e_{\text{min}}, \ldots, e_{\text{max}}\}
  \]

- Definition
  \[
  c = 1 \quad \text{if} \quad m_x \geq m_y, \quad \text{and} \quad c = 0 \quad \text{otherwise}
  \]
Range reduction of division

- Express the exact result $r = x/y$ as:

$$r = \ell \cdot 2^d \quad \Rightarrow \quad \text{RN}(x/y) = \text{RN}(\ell) \cdot 2^d$$

with

$$\ell \in [1, 2) \quad \text{and} \quad d \in \{e_{\text{min}}, \ldots, e_{\text{max}}\}$$

- Definition

$$c = 1 \quad \text{if} \quad m_x \geq m_y, \quad \text{and} \quad c = 0 \quad \text{otherwise}$$

- Range reduction

$$\frac{x}{y} = \left(2^{1-c} \cdot \frac{m_x}{m_y}\right) \cdot 2^d \quad \text{with} \quad d = e_x - e_y - 1 + c$$

$$:= \ell \in [1,2)$$
Range reduction of division

- Express the exact result $r = x/y$ as:

$$r = \ell \cdot 2^d \Rightarrow \text{RN}(x/y) = \text{RN}(\ell) \cdot 2^d$$

with

$$\ell \in [1, 2) \quad \text{and} \quad d \in \{e_{\min}, \ldots, e_{\max}\}$$

- Definition

$$c = 1 \quad \text{if} \quad m_x \geq m_y, \quad \text{and} \quad c = 0 \quad \text{otherwise}$$

- Range reduction

$$x/y = (2^{1-c} \cdot m_x/m_y) \cdot 2^d \quad \text{with} \quad d = e_x - e_y - 1 + c$$

$$: = \ell \in [1,2)$$

How to compute the correctly-rounded significand $\text{RN}(\ell)$?
Methods for computing the correctly-rounded significand

- **Iterative methods**: restoring, non-restoring, SRT, ...
  - Oberman and Flynn (1997)
  - minimal ILP exposure, sequential algorithm
Methods for computing the correctly-rounded significand

- **Iterative methods**: restoring, non-restoring, SRT, ...
  - Oberman and Flynn (1997)
  - minimal ILP exposure, sequential algorithm

- **Multiplicative methods**: Newton-Raphson, Goldschmidt
  - Piñeiro and Bruguera (2002) – Raina’s Ph.D., FLIP 0.3 (2006)
  - exploit available multipliers, more ILP exposure
Methods for computing the correctly-rounded significand

- **Iterative methods**: restoring, non-restoring, SRT, ...
  - Oberman and Flynn (1997)
  - minimal ILP exposure, sequential algorithm

- **Multiplicative methods**: Newton-Raphson, Goldschmidt
  - Piñeiro and Bruguera (2002) – Raina’s Ph.D., FLIP 0.3 (2006)
  - exploit available multipliers, more ILP exposure

- **Polynomial-based methods**
  - Agarwal, Gustavson and Schmookler (1999)
    → univariate polynomial evaluation
  - Our approach
    → bivariate polynomial evaluation: maximal ILP exposure
Correct rounding via truncated one-sided approximation

- How to compute $\text{RN}(\ell)$, with $\ell = 2^{1-c} \cdot m_x / m_y$?

- Three steps for correct rounding computation
  1. compute $v = 1.v_1 \ldots v_{k-2}$ such that $-2^{-p} \leq \ell - v < 0$
     → implied by $|(\ell + 2^{-p-1}) - v| < 2^{-p-1}$
     → bivariate polynomial evaluation
  2. compute $u$ as the truncation of $v$ after $p$ fraction bits
  3. determine $\text{RN}(\ell)$ after possibly adding $2^{-p}$
Correct rounding via truncated one-sided approximation

- How to compute $\text{RN}(\ell)$, with $\ell = 2^{1-c} \cdot m_x/m_y$?

- **Three steps** for correct rounding computation
  1. compute $v = 1.v_1 \ldots v_{k-2}$ such that $-2^{-p} \leq \ell - v < 0$
     → implied by $|(\ell + 2^{-p-1}) - v| < 2^{-p-1}$
     → bivariate polynomial evaluation
  2. compute $u$ as the truncation of $v$ after $p$ fraction bits
  3. determine $\text{RN}(\ell)$ after possibly adding $2^{-p}$

How to compute the one-sided approximation $v$ and then deduce $\text{RN}(\ell)$?
One-sided approximation via bivariate polynomials

1. Consider \( \ell + 2^{-p-1} \) as the exact result of the function

\[
F(s, t) = s/(1 + t) + 2^{-p-1}
\]

at the points \( s^* = 2^{1-c} \cdot m_x \) and \( t^* = m_y - 1 \)
One-sided approximation via bivariate polynomials

1. Consider \( \ell + 2^{-p-1} \) as the exact result of the function

\[
F(s, t) = s/(1 + t) + 2^{-p-1}
\]

at the points \( s^* = 2^{1-c} \cdot m_x \) and \( t^* = m_y - 1 \)

2. Approximate \( F(s, t) \) by a bivariate polynomial \( P(s, t) \)

\[
P(s, t) = s \cdot a(t) + 2^{-p-1}
\]

\( \rightarrow a(t) \): univariate polynomial approximant of \( 1/(1 + t) \)

\( \rightarrow \) approximation error \( E_{\text{approx}} \)
One-sided approximation via bivariate polynomials

1. Consider $\ell + 2^{-p-1}$ as the exact result of the function

   $$F(s, t) = s/(1 + t) + 2^{-p-1}$$

   at the points $s^* = 2^{1-c} \cdot m_x$ and $t^* = m_y - 1$

2. Approximate $F(s, t)$ by a bivariate polynomial $P(s, t)$

   $$P(s, t) = s \cdot a(t) + 2^{-p-1}$$

   $\rightarrow$ $a(t)$: univariate polynomial approximant of $1/(1 + t)$
   $\rightarrow$ approximation error $E_{\text{approx}}$

3. Evaluate $P(s, t)$ by a well-chosen efficient evaluation program $\mathcal{P}$

   $$v = \mathcal{P}(s^*, t^*)$$

   $\rightarrow$ evaluation error $E_{\text{eval}}$
One-sided approximation via bivariate polynomials

1. Consider $\ell + 2^{-p-1}$ as the exact result of the function

$$F(s, t) = s/(1 + t) + 2^{-p-1}$$

at the points $s^* = 2^{1-c} \cdot mx$ and $t^* = my - 1$

2. Approximate $F(s, t)$ by a bivariate polynomial $P(s, t)$

$$P(s, t) = s \cdot a(t) + 2^{-p-1}$$

$\rightarrow$ $a(t)$: univariate polynomial approximant of $1/(1 + t)$

$\rightarrow$ approximation error $E_{\text{approx}}$

3. Evaluate $P(s, t)$ by a well-chosen efficient evaluation program $\mathcal{P}$

$$v = \mathcal{P}(s^*, t^*)$$

$\rightarrow$ evaluation error $E_{\text{eval}}$

How to ensure that $|(\ell + 2^{-p-1}) - v| < 2^{-p-1}$?
Sufficient error bounds

To ensure \( |(\ell + 2^{-p-1}) - v| < 2^{-p-1} \)

it suffices to ensure that \( \mu \cdot E_{\text{approx}} + E_{\text{eval}} < 2^{-p-1} \),

since

\[
|\ell + 2^{-p-1} - v| \leq \mu \cdot E_{\text{approx}} + E_{\text{eval}} \quad \text{with} \quad \mu = 4 - 2^{3-p}
\]
Sufficient error bounds

- To ensure \(|(\ell + 2^{-p-1}) - v| < 2^{-p-1}\)

  it suffices to ensure that \(\mu \cdot E_{\text{approx}} + E_{\text{eval}} < 2^{-p-1}\),

  since

  \[|(\ell + 2^{-p-1}) - v| \leq \mu \cdot E_{\text{approx}} + E_{\text{eval}}\]

  with \(\mu = 4 - 2^{3-p}\)

- This gives the following sufficient conditions

  \[E_{\text{approx}} < 2^{-p-1}/\mu \implies E_{\text{eval}} < 2^{-p-1} - \mu \cdot E_{\text{approx}}\]
Sufficient error bounds

- To ensure \(|(\ell + 2^{-p-1}) - v| < 2^{-p-1}\)

  it suffices to ensure that \(\mu \cdot E_{\text{approx}} + E_{\text{eval}} < 2^{-p-1}\),

  since

  \[
  |(\ell + 2^{-p-1}) - v| \leq \mu \cdot E_{\text{approx}} + E_{\text{eval}} \quad \text{with} \quad \mu = 4 - 2^{3-p}
  \]

- This gives the following sufficient conditions

  \[
  E_{\text{approx}} \leq \theta \quad \text{with} \quad \theta < 2^{-p-1}/\mu \quad \Rightarrow \quad E_{\text{eval}} < \eta = 2^{-p-1} - \mu \cdot \theta
  \]
Example for the *binary32* division

- Sufficient conditions with $\mu = 4 - 2^{-21}$

\[
E_{\text{approx}} \leq \theta \quad \text{with} \quad \theta < 2^{-25}/\mu \quad \text{and} \quad E_{\text{eval}} < \eta = 2^{-25} - \mu \cdot \theta
\]
Example for the *binary32* division

- **Sufficient conditions with** $\mu = 4 - 2^{-21}$

\[ E_{\text{approx}} \leq \theta \quad \text{with} \quad \theta < 2^{-25}/\mu \quad \text{and} \quad E_{\text{eval}} < \eta = 2^{-25} - \mu \cdot \theta \]

- **Approximation of** $1/(1 + t)$ **by a Remez-like polynomial** of degree 10

\[
\begin{align*}
E_{\text{approx}} & \leq \theta, \\
\text{with} \quad \theta &= 3 \cdot 2^{-29} \approx 6 \cdot 10^{-9} \\
E_{\text{eval}} & < \eta, \\
\text{with} \quad \eta & \approx 7.4 \cdot 10^{-9}
\end{align*}
\]
Flowchart for generating efficient and certified C codes

\[ F(s,t) \quad E_{\text{approx}} \leq \theta \quad E_{\text{eval}} < \eta \]

Computation of polynomial approximant

Computation of low latency parenthesizations

Selection of effective parenthesizations

\[ |(\ell + 2^{-p-1}) - v| < 2^{-p-1} \]

ST231 features

C code

Certificate

v

truncation

u
Rounding condition: definition

- Approximation $u$ of $\ell$ with

$$\ell = 2^{1-c} \cdot m_x / m_y$$

- The exact value $\ell$ may have an infinite number of bits
  $\rightarrow$ the sticky bit cannot always be computed
Rounding condition: definition

- Approximation $u$ of $\ell$ with
  \[
  \ell = 2^{1-c} \cdot m_x / m_y
  \]
- The exact value $\ell$ may have an infinite number of bits
  $\Rightarrow$ the sticky bit cannot always be computed

- Compute $\text{RN}(\ell)$ requires to be able to decide whether $u \geq \ell$
  $\Rightarrow$ $\ell$ cannot be a midpoint
Rounding condition: definition

- Approximation \( u \) of \( \ell \) with
  \[
  \ell = 2^{1-c} \cdot \frac{m_x}{m_y}
  \]

- The exact value \( \ell \) may have an infinite number of bits
  → the sticky bit cannot always be computed

- Compute \( \text{RN}(\ell) \) requires to be able to decide whether \( u \geq \ell \)
  → \( \ell \) cannot be a midpoint

- Rounding condition: \( u \geq \ell \)
  \[
  u \geq \ell \iff u \cdot m_y \geq 2^{1-c} \cdot m_x
  \]
Rounding condition: implementation in integer arithmetic

- Rounding condition: \( u \cdot m_y \geq 2^{1-c} \cdot m_x \)

- Approximation \( u \) and \( m_y \): representable with 32 bits

\[
\begin{array}{c}
\text{ } \times \\
\hline
u \\
m_y \\
\hline
u \cdot m_y
\end{array}
\]

- \( u \cdot m_y \) is exactly representable with 64 bits
Rounding condition: implementation in integer arithmetic

- Rounding condition: \( u \cdot m_y \geq 2^{1-c} \cdot m_x \)

- Approximation \( u \) and \( m_y \): representable with 32 bits

\[
\begin{array}{c}
    u \\
\times \\
    m_y \\
\hline \\
    u \cdot m_y \\
2^{1-c} \cdot m_x \\
\end{array}
\]

- \( u \cdot m_y \) is exactly representable with 64 bits
- \( 2^{1-c} \cdot m_x \) is representable with 32 bits since \( c \in \{0, 1\} \)
Rounding condition: implementation in integer arithmetic

- **Rounding condition**: \( u \cdot m_y \geq 2^{1-c} \cdot m_x \)

- **Approximation** \( u \) and \( m_y \): representable with 32 bits

\[
\begin{align*}
\text{u} &\times \\
m_y &\geq \\
2^{1-c} \cdot m_x
\end{align*}
\]

- \( u \cdot m_y \) is exactly representable with 64 bits
- \( 2^{1-c} \cdot m_x \) is representable with 32 bits since \( c \in \{0, 1\} \)

\( \Rightarrow \) one \( 32 \times 32 \rightarrow 32\)-bit multiplication and one comparison
Flowchart for generating efficient and certified C codes

\[ F(s,t) \quad E_{\text{approx}} \leq \theta \quad E_{\text{eval}} < \eta \]

- Computation of polynomial approximant
- Computation of low latency parenthesizations
- Selection of effective parenthesizations

C code
Certificate

ST231 features
Flowchart for generating efficient and certified C codes

\[ F(s, t) \quad E_{\text{approx}} \leq \theta \quad E_{\text{eval}} < \eta \]

- Computation of polynomial approximant
- Computation of low latency parenthesizations
- Selection of effective parenthesizations
- C code
- Certificate

ST231 features
Outline of the talk

1. Design and implementation of floating-point operators

2. Low latency parenthesization computation
   - Classical evaluation methods
   - Computation of all parenthesizations
   - Towards low evaluation latency

3. Selection of effective evaluation parenthesizations

4. Numerical results

5. Conclusions

6. And now in ParLab: Debugging of floating-point programs
Objectives

- Compute an efficient parenthesization for evaluating $P(s, t)$
  - reduces the evaluation latency on unbounded parallelism
Objectives

- Compute an efficient parenthesization for evaluating $P(s, t)$
  $\rightarrow$ reduces the evaluation latency on unbounded parallelism

- Evaluation program $P = \text{main part of the full software implementation}$
  $\rightarrow$ dominates the cost
Objectives

- Compute an efficient parenthesization for evaluating \( P(s, t) \)
  \( \rightarrow \) reduces the evaluation latency on unbounded parallelism

- Evaluation program \( P \) = main part of the full software implementation
  \( \rightarrow \) dominates the cost

- Two families of algorithms
  - algorithms with coefficient adaptation: Knuth and Eve (60’s), Paterson and Stockmeyer (1964), ...
    \( \rightarrow \) ill-suited in the context of fixed-point arithmetic
  - algorithms without coefficient adaptation
Objectives

- Compute an efficient parenthesization for evaluating $P(s, t)$
  - reduces the evaluation latency on unbounded parallelism

- Evaluation program $P$ = main part of the full software implementation
  - dominates the cost

- Two families of algorithms
  - algorithms with coefficient adaptation: Knuth and Eve (60's), Paterson and Stockmeyer (1964), ...
    - ill-suited in the context of fixed-point arithmetic
  - algorithms without coefficient adaptation
Classical parenthesesizations for binary32 division

\[ P(s, t) = 2^{-25} + s \cdot \sum_{0 \leq i \leq 10} a_i \cdot t^i \]

- Horner’s rule: \((3 + 1) \times 11 = 44\) cycles
  - \(\rightarrow\) no ILP exposure
Classical parenthesizations for *binary32* division

\[ P(s, t) = 2^{-25} + s \cdot \sum_{0 \leq i \leq 10} a_i \cdot t^i \]

- Horner’s rule: \((3 + 1) \times 11 = 44\) cycles
  \rightarrow \text{no ILP exposure}

- Second-order Horner’s rule: 27 cycles
  \rightarrow \text{evaluation of odd and even parts independently with Horner, more ILP}
Classical parentheses for *binary32* division

\[ P(s, t) = 2^{-25} + s \cdot \sum_{0 \leq i \leq 10} a_i \cdot t^i \]

- **Horner’s rule:** \((3 + 1) \times 11 = 44\) cycles
  \(\rightarrow\) no ILP exposure

- **Second-order Horner’s rule:** 27 cycles
  \(\rightarrow\) evaluation of odd and even parts independently with Horner, more ILP

- **Estrin’s method:** 19 cycles
  \(\rightarrow\) evaluation of high and low parts in parallel, even more ILP
  \(\rightarrow\) distributing the multiplication by \(s\) in the evaluation of \(a(t)\) \(\rightarrow\) 16 cycles
Classical parenthesizations for *binary32* division

\[ P(s, t) = 2^{-25} + s \cdot \sum_{0 \leq i \leq 10} a_i \cdot t^i \]

- **Horner’s rule:** \((3 + 1) \times 11 = 44\) cycles
  - \(\rightarrow\) no ILP exposure
- **Second-order Horner’s rule:** 27 cycles
  - \(\rightarrow\) evaluation of odd and even parts independently with *Horner*, more ILP
- **Estrin’s method:** 19 cycles
  - \(\rightarrow\) evaluation of high and low parts in parallel, even more ILP
  - \(\rightarrow\) distributing the multiplication by \(s\) in the evaluation of \(a(t) \rightarrow 16\) cycles

- ... 

  *We can do better.*

How to explore the solution space of parenthesizations?
Algorithm for computing all parenthesizations

\[ a(x, y) = \sum_{0 \leq i \leq n_x} \sum_{0 \leq j \leq n_y} a_{i,j} \cdot x^i \cdot y^j \quad \text{with} \quad n = n_x + n_y, \quad \text{and} \quad a_{n_x,n_y} \neq 0 \]

Example

Let \( a(x, y) = a_{0,0} + a_{1,0} \cdot x + a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y \). Then

\[ a_{1,0} + a_{1,1} \cdot y \] is a valid expression, while \( a_{1,0} \cdot x + a_{1,1} \cdot x \) is not.
Algorithm for computing all parenthesizations

\[ a(x, y) = \sum_{0 \leq i \leq n_x} \sum_{0 \leq j \leq n_y} a_{i,j} \cdot x^i \cdot y^j \quad \text{with} \quad n = n_x + n_y, \quad \text{and} \quad a_{n_x,n_y} \neq 0 \]

Example

Let \( a(x, y) = a_{0,0} + a_{1,0} \cdot x + a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y \). Then

\[ a_{1,0} + a_{1,1} \cdot y \] is a valid expression, while \( a_{1,0} \cdot x + a_{1,1} \cdot x \) is not.

- Exhaustive algorithm: iterative process
  \[ \rightarrow \text{step } k = \text{computation of all the valid expressions of total degree } k \]

- 3 building rules for computing all parenthesizations
Low latency parenthesization computation

Computation of all parenthesizations

Number of parenthesizations

<table>
<thead>
<tr>
<th>$n_x = 1$</th>
<th>$n_x = 2$</th>
<th>$n_x = 3$</th>
<th>$n_x = 4$</th>
<th>$n_x = 5$</th>
<th>$n_x = 6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_y = 0$</td>
<td>1</td>
<td>7</td>
<td>163</td>
<td>11602</td>
<td>2334244</td>
</tr>
<tr>
<td>$n_y = 1$</td>
<td>51</td>
<td>67467</td>
<td>1133220387</td>
<td>207905478247998</td>
<td>...</td>
</tr>
<tr>
<td>$n_y = 2$</td>
<td>67467</td>
<td>106191222651</td>
<td>10139277122276921118</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Number of generated parenthesizations for evaluating a bivariate polynomial

- Timings for parenthesization computation
  - for univariate polynomial of degree 5 $\approx 1$h on a 2.4 GHz core
  - for bivariate polynomial of degree (2,1) $\approx 30$s
  - for $P(s, t)$ of degree (3,1) $\approx 7$s (88384 schemes)

- Optimization for univariate polynomial and $P(s, t)$
  - univariate polynomial of degree 5 $\approx 4$min
  - for $P(s, t)$ of degree (3,1) $\approx 2$s (88384 schemes)
Number of parenthesizations

→ minimal latency for univariate polynomial of degree 5: 10 cycles (36 schemes)
→ minimal latency for univariate polynomial of degree 5: 10 cycles (36 schemes)

How to compute only parenthesizations of low latency?
Determination of a *target* latency

- Target latency = *minimal cost* for evaluating

\[ a_{0,0} + a_{nx,ny} \cdot x^{nx} y^{ny} \]

- if no scheme satisfies \( \tau \) then increase \( \tau \) and restart
Determination of a \textit{target} latency

- Target latency $= \text{minimal cost for evaluating}$

\[ a_{0,0} + a_{n_x,n_y} \cdot x^{n_x} y^{n_y} \]

- if no scheme satisfies $\tau$ then increase $\tau$ and restart

- Static target latency $\tau_{\text{static}}$
  - as general as evaluating $a_{0,0} + x^{n_x+n_y+1}$

\[ \tau_{\text{static}} = A + M \times \lceil \log_2(n_x + n_y + 1) \rceil \]
Determination of a target latency

- **Target latency** = minimal cost for evaluating
  \[ a_{0,0} + a_{n_x,n_y} \cdot x^{n_x} y^{n_y} \]
  - if no scheme satisfies \( \tau \) then increase \( \tau \) and restart

- **Static target latency** \( \tau_{static} \)
  - as general as evaluating \( a_{0,0} + x^{n_x+n_y+1} \)
  \[ \tau_{static} = A + M \times \lceil \log_2(n_x + n_y + 1) \rceil \]

- **Dynamic target latency** \( \tau_{dynamic} \)
  - cost of operator on \( a_{n_x,n_y} \) and delay on intederminates
  - dynamic programming
Optimized search of *best* parenthesizations

Example

Let \( a(x, y) \) be a degree-2 bivariate polynomial

\[
a(x, y) = a_{0,0} + a_{1,0} \cdot x + a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y.
\]

\[ \Rightarrow \text{find a best splitting of the polynomial} \rightarrow \text{low latency} \]
Optimized search of *best* parenthesizations

Example

Let \( a(x, y) \) be a degree-2 bivariate polynomial

\[
a(x, y) = a_{0,0} + a_{1,0} \cdot x + a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y.
\]

\[\Rightarrow \text{find a best splitting of the polynomial} \rightarrow \text{low latency}\]

\[
\left( a_{0,0} + a_{1,0} \cdot x + a_{0,1} \cdot y \right) + \left( a_{1,1} \cdot x \cdot y \right)
\]
Optimized search of best parenthesizations

Example
Let $a(x, y)$ be a degree-2 bivariate polynomial

$$a(x, y) = a_{0,0} + a_{1,0} \cdot x + a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y.$$ 

$\Rightarrow$ find a best splitting of the polynomial $\rightarrow$ low latency

$$(a_{0,0} + a_{1,0} \cdot x) + a_{0,1} \cdot y + (a_{1,1} \cdot x \cdot y)$$
Optimized search of *best* parenthesizations

Example
Let $a(x, y)$ be a degree-2 bivariate polynomial

$$a(x, y) = a_{0,0} + a_{1,0} \cdot x + a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y.$$  

⇒ find a best splitting of the polynomial → low latency

$$\left( a_{0,0} + (a_{1,0} \cdot x + a_{0,1} \cdot y) \right) + \left( a_{1,1} \cdot x \cdot y \right)$$
Optimized search of *best* parenthesizations

Example

Let \( a(x, y) \) be a degree-2 bivariate polynomial

\[
a(x, y) = a_{0,0} + a_{1,0} \cdot x + a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y.
\]

⇒ find a best splitting of the polynomial → low latency

\[
\left( a_{0,0} + a_{1,0} \cdot x \right) + \left( a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y \right)
\]
Optimized search of *best* parenthesizations

Example

Let \( a(x, y) \) be a degree-2 bivariate polynomial

\[
a(x, y) = a_{0,0} + a_{1,0} \cdot x + a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y.
\]

⇒ find a best splitting of the polynomial → low latency

\[
a_{0,0} + \left( a_{1,0} \cdot x + a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y \right)
\]
Optimized search of best parenthesizations

Example

Let $a(x, y)$ be a degree-2 bivariate polynomial

$$a(x, y) = a_{0,0} + a_{1,0} \cdot x + a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y.$$
Efficient evaluation parenthesization generation

\[ P(s, t) = 2^{-25} + s \cdot \sum_{0 \leq i \leq 10} a_i \cdot t^i \]

- First target latency \( \tau = 13 \)
  - \( \rightarrow \) no parenthesization found
Efficient evaluation parenthesization generation

\[ P(s, t) = 2^{-25} + s \cdot \sum_{0 \leq i \leq 10} a_i \cdot t^i \]

- First target latency \( \tau = 13 \)
  \( \rightarrow \) no parenthesization found

- Second target latency \( \tau = 14 \)
  \( \rightarrow \) obtained in about 10 sec.

- Classical methods
  - Horner: 44 cycles,
  - Estrin: 19 cycles,
  - Estrin by distributing \( s \): 16 cycles
Flowchart for generating efficient and certified C codes

\[ F(s,t) \quad E_{\text{approx}} \leq \theta \quad E_{\text{eval}} < \eta \]

- Computation of polynomial approximant
- Computation of low latency parenthesizations
- Selection of effective parenthesizations
- ST231 features
- C code
- Certificate
Flowchart for generating efficient and certified C codes

\[ F(s,t) \quad E_{\text{approx}} \leq \theta \quad E_{\text{eval}} < \eta \]

Computation of polynomial approximant

Computation of low latency parenthesizations

Selection of effective parenthesizations

C code

Certificate

ST231 features
Outline of the talk

1. Design and implementation of floating-point operators

2. Low latency parenthesization computation

3. Selection of effective evaluation parenthesizations
   - General framework
   - Automatic certification of generated C codes

4. Numerical results

5. Conclusions

6. And now in ParLab: Debugging of floating-point programs
Selection of effective parenthesizations

1. Arithmetic Operator Choice
   ▶ all intermediate variables are of constant sign
Selection of effective parenthesizations

1. Arithmetic Operator Choice
   ▶ all intermediate variables are of constant sign

2. Scheduling on a simplified model of the ST231
   ▶ constraints of architecture: cost of operators, instructions bundling, ...
   ▶ delays on indeterminates
Selection of effective evaluation parenthesizations

1. Arithmetic Operator Choice
   ▶ all intermediate variables are of constant sign

2. Scheduling on a simplified model of the ST231
   ▶ constraints of architecture: cost of operators, instructions bundling, ...
   ▶ delays on indeterminates

3. Certification of generated C code
   ▶ **straightline** polynomial evaluation program
   ▶ “certified C code”: we can bound the evaluation error in integer arithmetic
Certification of evaluation error for binary32 division

- Sufficient conditions with $\mu = 4 - 2^{-21}$

\[ E_{\text{approx}} \leq \theta \quad \text{with} \quad \theta < 2^{-25}/\mu \quad \text{and} \quad E_{\text{eval}} < \eta = 2^{-25} - \mu \cdot \theta \]

- $E_{\text{approx}} \leq \theta$, with $\theta = 3 \cdot 2^{-29} \approx 6 \cdot 10^{-9}$

- $E_{\text{eval}} < \eta$, with $\eta \approx 7.4 \cdot 10^{-9}$
Certification of evaluation error for *binary32* division

- **Case 1**: $m_x \geq m_y$ → condition satisfied
- **Case 2**: $m_x < m_y$ → condition not satisfied: $E_{\text{eval}} \geq \eta$

$s^* = 3.935581684112548828125$ and $t^* = 0.97490441799163818359375$
Certification of evaluation error for *binary32* division

- **Case 1:** $m_x \geq m_y \rightarrow$ condition satisfied
- **Case 2:** $m_x < m_y \rightarrow$ condition not satisfied: $E_{\text{eval}} \geq \eta$

$$s^* = 3.935581684112548828125 \text{ and } t^* = 0.97490441799163818359375$$

1. determine an interval $I$ around this point
Certification of evaluation error for *binary32* division

- **Case 1:** $m_x \geq m_y \rightarrow$ condition satisfied
- **Case 2:** $m_x < m_y \rightarrow$ condition not satisfied: $E_{\text{eval}} \geq \eta$

\[ s^* = 3.935581684112548828125 \text{ and } t^* = 0.97490441799163818359375 \]

1. determine an interval $I$ around this point
2. compute $E_{\text{approx}}$ over $I$
3. determine an evaluation error bound $\eta$
4. check if $E_{\text{eval}} < \eta$?
Certification of evaluation error for *binary32* division

- Sufficient conditions for each subinterval, with $\mu = 4 - 2^{-21}$

\[
E_{\text{approx}}^{(i)} \leq \theta^{(i)} \quad \text{with} \quad \theta^{(i)} < 2^{-25}/\mu \quad \text{and} \quad E_{\text{eval}}^{(i)} < \eta^{(i)} = 2^{-25} - \mu \cdot \theta^{(i)}
\]
Certification of evaluation error for *binary32* division

- Sufficient conditions for each subinterval, with $\mu = 4 - 2^{-21}$

\[
E_{\text{approx}}^{(i)} \leq \theta^{(i)} \quad \text{with} \quad \theta^{(i)} < 2^{-25}/\mu \quad \text{and} \quad E_{\text{eval}}^{(i)} < \eta^{(i)} = 2^{-25} - \mu \cdot \theta^{(i)}
\]

- $E_{\text{approx}}^{(i)} \leq \theta^{(i)}$  
- $E_{\text{eval}}^{(i)} < \eta^{(i)}$
Certification using a dichotomy-based strategy

- Implementation of the splitting by **dichotomy**

  - for each $\mathcal{T}^{(i)}$
    1. compute a certified approximation error bound $\theta^{(i)}$
    2. determine an evaluation error bound $\eta^{(i)}$
    3. check this bound: $E_{\text{eval}}^{(i)} < \eta^{(i)}$

  $\Rightarrow$ if this bound is not satisfied, $\mathcal{T}^{(i)}$ is split up into 2 subintervals
Certification using a dichotomy-based strategy

- Implementation of the splitting by dichotomy

  - for each $\mathcal{T}^{(i)}$
    1. compute a certified approximation error bound $\theta^{(i)}$
    2. determine an evaluation error bound $\eta^{(i)}$
    3. check this bound: $E_{\text{eval}}^{(i)} < \eta^{(i)}$

$\Rightarrow$ if this bound is not satisfied, $\mathcal{T}^{(i)}$ is split up into 2 subintervals
Certification using a dichotomy-based strategy

- Implementation of the splitting by dichotomy
  
  ▶ for each $\mathcal{T}^{(i)}$
    
    1. compute a certified approximation error bound $\theta^{(i)}$
    
    2. determine an evaluation error bound $\eta^{(i)}$
    
    3. check this bound: $E_{\text{eval}}^{(i)} < \eta^{(i)}$

  $\Rightarrow$ if this bound is not satisfied, $\mathcal{T}^{(i)}$ is split up into 2 subintervals

- Example of *binary32* implementation
  
  → launched on a 64 processor grid
  
  → 36127 subintervals found in several hours ($\approx 5$h.)
Outline of the talk

1. Design and implementation of floating-point operators
2. Low latency parenthesization computation
3. Selection of effective evaluation parenthesizations
4. Numerical results
5. Conclusions
6. And now in ParLab: Debugging of floating-point programs
Performances of FLIP on ST231

Performances on ST231, in RoundTiesToEven

⇒ Speed-up between 20 and 50%
Performances of FLIP on ST231

⇒ Speed-up between 20 and 50 %

- Implementations of other operators

<table>
<thead>
<tr>
<th>operator</th>
<th>$x^{-1}$</th>
<th>$x^{-1/2}$</th>
<th>$x^{1/3}$</th>
<th>$x^{-1/3}$</th>
<th>$x^{-1/4}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25</td>
<td>29</td>
<td>34</td>
<td>40</td>
<td>42</td>
</tr>
</tbody>
</table>

Performances on ST231, in RoundTiesToEven (in number of cycles)
Outline of the talk

1. Design and implementation of floating-point operators
2. Low latency parenthesization computation
3. Selection of effective evaluation parenthesizations
4. Numerical results
5. Conclusions
6. And now in ParLab: Debugging of floating-point programs
Conclusions

- Design and implementation of floating-point operators
  - uniform approach for correctly-rounded roots and their reciprocals
  - extension to correctly-rounded division
Conclusions

- Design and implementation of floating-point operators
  - uniform approach for correctly-rounded roots and their reciprocals
  - extension to correctly-rounded division
  - polynomial evaluation-based method, very high ILP exposure
- $\Rightarrow$ new, much faster version of FLIP
Conclusions

- Design and implementation of floating-point operators
  - uniform approach for correctly-rounded roots and their reciprocals
  - extension to correctly-rounded division
  - polynomial evaluation-based method, very high ILP exposure
  ⇒ new, much faster version of FLIP

- Code generation for efficient and certified polynomial evaluation
  - methodologies and tools for automating polynomial evaluation implementation
  - heuristics and techniques for generating quickly efficient and certified C codes
  ⇒ CGPE: allows to write and certify automatically ≈ 50 % of the codes of FLIP
Outline of the talk

1. Design and implementation of floating-point operators
2. Low latency parenthesization computation
3. Selection of effective evaluation parenthesizations
4. Numerical results
5. Conclusions
6. And now in ParLab: Debugging of floating-point programs
Debugging of floating-point programs

- Tool for detecting and remedying anomalies in floating-point programs
  - either at C code level or at run-time

- What are the usual anomalies?
  - rounding error accumulations
  - conditional branches involving floating-point comparisons
    - may fail due to the subtleties of floating-point arithmetic
  - difficulties of programming languages
    - Fortran: constants converted in full double precision accuracy if written with the $dX$ notation
  - overflows, resolution of ill-conditioned problems
    - returned result may be completely wrong
  - benign / catastrophic cancellation, ...
Debugging of floating-point programs

- Tool for detecting and remedying anomalies in floating-point programs
  - either at C code level or at run-time

- How to detect these usual anomalies?
  - altering rounding mode of floating-point arithmetic hardware
    - may not be used for remedying problems
  - extending precision of floating-point computation
    - run time may increase significantly (due to the use of software interface)
  - using interval arithmetic
    - produces a certificate, but run time cost is the greatest
Debugging of floating-point programs

Tool for detecting and remedying anomalies in floating-point programs

→ either at C code level or at run-time

How to detect these usual anomalies?

▶ altering rounding mode of floating-point arithmetic hardware
  → may not be used for remedying problems

▶ extending precision of floating-point computation
  → run time may increase significantly (due to the use of software interface)

▶ using interval arithmetic
  → produces a certificate, but run time cost is the greatest

How to detect quickly the most sensitive part of a C program?
Detection using *delta-debugging*

- **Principle**: find a minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, double precision, ...)
  
  → implementation by binary search

```c
#include <math.h>
#include <stdio.h>

int main( void )
{
  float a = 1e15f;
  float b = 1.0f;
  float c = a + b;
  float d = c - a;   // d = 0.0

  printf("The value of d is: %1.19e\n", d);
  return 0;
}
```

- **What is the value of \( d \)?**
  - Using *binary32* floating-point arithmetic
    
    → \( d = 0.0 \)
  - Using *binary64* floating-point arithmetic
    
    → \( d = 1.0 \)
Detection using \textit{delta-debugging}

- **Principle**: find a minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, double precision, ...)
  
  $\rightarrow$ implementation by binary search

```c
#include <math.h>
#include <stdio.h>

int main( void )
{
    double a = 1e15f;
    double b = 1.0f;
    float c = a + b;
    float d = c - a;  // $d = 0.0$

    printf("The value of d is: %1.19e\n", d);
    return 0;
}
```

- **What is the value of $d$?**
  - Using \textit{binary32} floating-point arithmetic
    $\rightarrow$ $d = 0.0$
  - Using \textit{binary64} floating-point arithmetic
    $\rightarrow$ $d = 1.0$
Detection using *delta-debugging*

- **Principle**: find a minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, double precision, ...)

  → implementation by binary search

```c
#include <math.h>
#include <stdio.h>

int main( void )
{
    float a = 1e15f;
    float b = 1.0f;
    double c = a + b;
    double d = c - a;  // d = 0.0

    printf("The value of d is: %1.19e\n", d);
    return 0;
}
```

- **What is the value of \(d\)?**
  - Using *binary32* floating-point arithmetic
    → \( d = 0.0 \)
  - Using *binary64* floating-point arithmetic
    → \( d = 1.0 \)
Detection using \textit{delta-debugging}

\begin{itemize}
  \item \textbf{Principle}: find a minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, double precision, ...)
  \item \textarrow{implementation by binary search}
\end{itemize}

```c
#include <math.h>
#include <stdio.h>

int main( void )
{
  double a = 1e15f;
  float b = 1.0f;
  float c = a + b;
  float d = c - a;  // d = 0.0
  printf("The value of d is: %1.19e\n", d);
  return 0;
}
```

\begin{itemize}
  \item \textbf{What is the value of } $d$?
  \begin{itemize}
    \item Using \textit{binary32} floating-point arithmetic
      \item \textarrow{$d = 0.0$}
    \item Using \textit{binary64} floating-point arithmetic
      \item \textarrow{$d = 1.0$}
  \end{itemize}
\end{itemize}
Detection using \textit{delta-debugging}

- **Principle**: find a minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, double precision, ...)
  
  \[ \rightarrow \text{implementation by binary search} \]

```
#include <math.h>
#include <stdio.h>

int main( void ) {
    float a = 1e15f;    
    double b = 1.0f;    
    float c = a + b;    
    float d = c - a;    
    // d = 0.0

    printf("The value of d is: %1.19e\n", d);
    return 0;
}
```

- **What is the value of \(d\)?**
  - Using \textit{binary32} floating-point arithmetic
    
    \[ \rightarrow d = 0.0 \]
  - Using \textit{binary64} floating-point arithmetic
    
    \[ \rightarrow d = 1.0 \]
Detection using *delta-debugging*

- **Principle**: find a minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, double precision, ...)
  
  → implementation by binary search

```c
#include <math.h>
#include <stdio.h>

int main( void )
{
    float a = 1e15f;
    float b = 1.0f;
    double c = a + b;
    float d = c - a;   // d = 0.0

    printf("The value of d is: %1.19e\n", d);
    return 0;
}
```

- **What is the value of $d$?**
  - Using *binary32* floating-point arithmetic
    
    → $d = 0.0$
  - Using *binary64* floating-point arithmetic
    
    → $d = 1.0$
Detection using *delta-debugging*

- **Principle**: find a minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, double precision, ...)
  
  → implementation by binary search

```c
#include <math.h>
#include <stdio.h>

int main( void )
{
    float a = 1e15f;
    float b = 1.0f;
    float c = a + b;
    double d = c - a;  // d = 0.0

    printf("The value of d is: %1.19e\n", d);
    return 0;
}
```

- **What is the value of \(d\)?**
  
  - Using *binary32* floating-point arithmetic
    
    → \(d = 0.0\)
  
  - Using *binary64* floating-point arithmetic
    
    → \(d = 1.0\)
Detection using \textit{delta-debugging}

\textbf{Principle}: find a minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, double precision, ...)

\textarrow implementation by binary search

```c
#include <math.h>
#include <stdio.h>

int main( void )
{
    float a = 1e15f;
    double b = 1.0f;
    double c = a + b;
    double d = c - a;  // d = 1.0

    printf("The value of d is: %1.19e\n", d);
    return 0;
}
```

\textbf{What is the value of }$d$\textbf{?}

\begin{itemize}
    \item Using \textit{binary32} floating-point arithmetic
        \textarrow $d = 0.0$
    \item Using \textit{binary64} floating-point arithmetic
        \textarrow $d = 1.0$
\end{itemize}
Detection using *delta-debugging*

**Principle:** find a minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, double precision, ...)

→ implementation by binary search

```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    float a = 1e15f;
    double b = 1.0f;
    double c = a + b;
    float d = c - a; // d = 1.0

    printf("The value of d is: %1.19e\n", d);
    return 0;
}
```

**What is the value of $d$?**

- Using *binary32* floating-point arithmetic
  → $d = 0.0$

- Using *binary64* floating-point arithmetic
  → $d = 1.0$
Current work

- **Delta-debugging**
  - how to determine initial set of changes?
  - implementation of other transformations

- Implementation of an exception handler
  - may be useful for building initial set of *delta-debugging* algorithm

- Detection of infinite loops, ...
Implementation of binary floating-point arithmetic on embedded integer processors

Polynomial evaluation-based algorithms and certified code generation

Guillaume Revy
ParLab EECS University of California, Berkeley

Ph.D. thesis’ work done under the direction of Claude-Pierre Jeannerod and Gilles Villard
Arénaire INRIA project-team (LIP, Ens Lyon, France)