Implementation of binary floating-point arithmetic on embedded integer processors
Polynomial evaluation-based algorithms and certified code generation

Guillaume Revy
Advisors: Claude-Pierre Jeannerod and Gilles Villard

Arénaire INRIA project-team (LIP, Ens Lyon)  Université de Lyon  CNRS
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**
Overview of the ST231 architecture

- 4-issue VLIW 32-bit integer processor → no FPU
- Parallel execution unit
  - 4 integer ALU
  - 2 pipelined multipliers $32 \times 32 \rightarrow 32$
- Latencies: ALU → 1 cycle, Mul → 3 cycles
Overview of the ST231 architecture

- 4-issue VLIW 32-bit integer processor
  → no FPU
- Parallel execution unit
  ▶ 4 integer ALU
  ▶ 2 pipelined multipliers $32 \times 32 \rightarrow 32$
- Latencies: ALU → 1 cycle, Mul → 3 cycles

- VLIW (Very Long Instruction Word)
  → instructions grouped into bundles
  → Instruction-Level Parallelism (ILP) explicitly exposed by the compiler
Overview of the ST231 architecture

- 4-issue VLIW 32-bit integer processor
  → no FPU
- Parallel execution unit
  ▶ 4 integer ALU
  ▶ 2 pipelined multipliers $32 \times 32 \rightarrow 32$
- Latencies: ALU → 1 cycle, Mul → 3 cycles

VLIW (Very Long Instruction Word)
→ instructions grouped into bundles
→ Instruction-Level Parallelism (ILP) explicitly exposed by the compiler

```
uint32_t R1 = A0 + C;
uint32_t R2 = A3 * X;
uint32_t R3 = A1 * X;
uint32_t R4 = X * X;
```

<table>
<thead>
<tr>
<th></th>
<th>Issue 1</th>
<th>Issue 2</th>
<th>Issue 3</th>
<th>Issue 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R1</td>
<td>R2</td>
<td></td>
<td>R3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>R4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 $\mu$Clibc, 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

- 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?*

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

  *In the particular case of polynomial evaluation, can we 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
  - no
  - yes
    - Floating-point number unpacking
      - function independent
      - function dependent
    - Normalization
    - Range reduction
      - Result sign/exponent computation
      - Result significand approximation
        - Rounding condition decision
        - Correct rounding computation
      - Result reconstruction
    - Special output selection

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

Objects

→ Low latency, correctly-rounded implementations
→ ILP exposure
Basic blocks for implementing correctly-rounded operators

- (X, Y)
- Special input detection
- No
- Floating-point number unpacking
- Normalization
- Range reduction
- Result sign/exponent computation
- Result reconstruction
- R

- Yes
- Special input detection
- Special output selection
- Special input detection

Algorithm:

1. Special input detection
2. Floating-point number unpacking
3. Normalization
4. Range reduction
5. Result sign/exponent computation
6. Result reconstruction
7. Special output selection

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

- ST231 features

- Uniform approach for nth roots and their reciprocals
  → polynomial evaluation
- Extension to division
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
5. C code
6. Certificate
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

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

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

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

Constraints

- Accuracy of approximant and C code
  - Sollya
  - Interval arithmetic (MPFI), Gappa
- Low evaluation latency on ST231, ILP exposure
  - ?

C code

Certificate
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 parenthesesization computation
   Classical evaluation methods
   Computation of all parenthesesizations
   Towards low evaluation latency

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

4. Numerical results

5. Conclusions and perspectives
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 and perspectives
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 $x$

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

  - $s_x$: 1 bit
  - $E_x$: $w = k - p$ bits
  - $T_x$: $p - 1$ bits

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

  $\rightarrow$ integer and fixed-point arithmetic
Notation and assumptions

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

\[
\begin{align*}
 & s_x & E_x = e_x - e_{\text{min}} - 1 & T_x = m_{x,1} \ldots m_{x,p-1} \\
 & 1 \text{ bit} & w = k - p \text{ bits} & p - 1 \text{ bits}
\end{align*}
\]

- **Computation**: $k$-bit unsigned integers
  \[\rightarrow\] integer and fixed-point arithmetic
Range reduction of division

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

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

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

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

- Definition

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

- Express the exact result \( r = x/y \) as:

\[
r = \ell \cdot 2^d \implies \text{RN}(x/y) = \text{RN}(\ell) \cdot 2^d
\]

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

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

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

- Express the exact result \( r = \frac{x}{y} \) as:

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

with

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

- Definition

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

- Range reduction

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

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

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

- **Three steps** for correct rounding computation
  1. compute $v = 1.\cdot v_1 \ldots v_{k-2}$ such that $-2^{-p} \leq \ell - v < 0$
    
    $\rightarrow$ implied by $|(\ell + 2^{-p-1}) - v| < 2^{-p-1}$
    
    $\rightarrow$ 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 $E_{approx}$

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

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

$\rightarrow$ evaluation error $E_{eval}$
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 $s^* = 2^{1-c} \cdot m_x$ 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 $P$

$$v = P(s^*, t^*)$$

$\rightarrow$ 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

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

- This gives the following sufficient conditions

  \[ E_{\text{approx}} < \frac{2^{-p-1}}{\mu} \quad \Rightarrow \quad E_{\text{eval}} < 2^{-p-1} - \mu \cdot E_{\text{approx}} \]
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

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

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

- 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

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

\[
E_{\text{eval}} < \eta, \\
\text{with} \quad \eta \approx 7.4 \cdot 10^{-9}
\]
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

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

C code Certificate

\( u \)

\( v \)

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
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
  $\rightarrow$ 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
  $\rightarrow$ 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: $u \geq \ell$
  \[
  u \geq \ell \iff u \cdot m_y \geq 2^{1-c} \cdot m_x
  \]

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

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

- \( u \cdot m_y \) 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

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

- \( u \cdot m_y \) 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

  \[
  u \cdot m_y \geq 2^{1-c} \cdot m_x
  \]

- \( u \cdot m_y \) 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
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
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 and perspectives
Objectives

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

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

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

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

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

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

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

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

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

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

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

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

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

... We can do better.

How to explore the solution space of parentheses?
Classical parentheses for \textit{binary32} division

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

- 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 parentheses for *binary32* division

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

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

... We can do better.

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

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

**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

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

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

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
Rules for building valid expressions

Consider step $k$ of the algorithm

- $E^{(k)}$: valid expressions of total degree $k$
- $P^{(k)}$: powers $x^i y^j$ of total degree $k = i + j$
Rules for building \textit{valid} expressions

Consider step $k$ of the algorithm

\begin{itemize}
\item $E^{(k)}$: valid expressions of total degree $k$
\item $P^{(k)}$: powers $x^i y^j$ of total degree $k = i + j$
\end{itemize}

Rule R1 for building the powers

\[
\deg(p) = \deg(p_1) + \deg(p_2)
\]

\[
\deg(p_1) \leq \left\lfloor \frac{k}{2} \right\rfloor \quad \left\lceil \frac{k}{2} \right\rceil \leq \deg(p_2) < k
\]
Rules for building valid expressions

Consider step $k$ of the algorithm

- $E^{(k)}$: valid expressions of total degree $k$
- $P^{(k)}$: powers $x^i y^j$ of total degree $k = i + j$

Rule R2 for expressions by multiplications

\[
\text{deg}(e) = \text{deg}(e') + \text{deg}(p)
\]

\[
\begin{align*}
\text{deg}(e') &< k \\
\text{deg}(p) &\leq k
\end{align*}
\]
Rules for building *valid* expressions

Consider step $k$ of the algorithm

- $E^{(k)}$: valid expressions of total degree $k$
- $P^{(k)}$: powers $x^i y^j$ of total degree $k = i + j$

Rule R3 for expressions by additions

$$\deg(e) = \max(\deg(e_1), \deg(e_2))$$

```
        +
       /\n      /  \n     e_1  e_2

\deg(e_1) = k \quad \deg(e_2) \leq k
```
Number of Parenthesizations

<table>
<thead>
<tr>
<th></th>
<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>
<td>1304066578</td>
</tr>
<tr>
<td>$n_y = 1$</td>
<td>51</td>
<td>67467</td>
<td>1133220387</td>
<td>207905478247998</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$n_y = 2$</td>
<td>67467</td>
<td>106191222651</td>
<td>10139277122276921118</td>
<td>...</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 $\approx$ 1h on a 2.4 GHz core
  - for bivariate polynomial of degree (2,1) $\approx$ 30s
  - for $P(s, t)$ of degree (3,1) $\approx$ 7s (88384 schemes)

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

→ 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

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

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

- Target latency = \textbf{minimal cost} for evaluating

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

\begin{itemize}
  \item if no scheme satisfies $\tau$ then increase $\tau$ and restart
\end{itemize}

- Static target latency $\tau_{\text{static}}$

\begin{itemize}
  \item as general as evaluating $a_{0,0} + x^{n_x+n_y+1}$
  \item $\tau_{\text{static}} = A + M \times \left\lceil \log_2(n_x + n_y + 1) \right\rceil$
\end{itemize}
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 \( \tau_{\text{static}} \)
  - as general as evaluating \( a_{0,0} + x^{n_x+n_y+1} \)
  \[ \tau_{\text{static}} = A + M \times \lceil \log_2(n_x + n_y + 1) \rceil \]

- Dynamic target latency \( \tau_{\text{dynamic}} \)
  - cost of operator on \( a_{n_x,n_y} \) and delay on indeterminates
  - dynamic programming
Determination of a \textit{target} latency

- Target latency = \textbf{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

**Example**

- Degree-9 bivariate polynomial: $n_x = 8$ and $n_y = 1$
- Latencies: $A = 1$ and $M = 3$
- Delay: $y$ available 9 cycles later than $x$

\[
\begin{array}{l|l}
\tau_{\text{static}} & \tau_{\text{dynamic}} \\
1 + 3 \times \lceil \log_2(10) \rceil = 13 \text{ cycles} & 16 \text{ cycles}
\end{array}
\]
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
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

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

⇒ 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

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

\( \Rightarrow \) find a best *splitting* of the polynomial \( \rightarrow \) 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

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

⇒ find a best splitting of the polynomial → low latency
Efficient evaluation parenthesization generation

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

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

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

- 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

C code
Certificate

ST231 features

Implementation of binary floating-point arithmetic on embedded integer processors
Flowchart for generating efficient and certified C codes

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

1. Computation of polynomial approximant
2. Computation of low latency parenthesizations
3. Selection of effective parenthesizations
4. ST231 features
5. C code
6. Certificate

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 and perspectives
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 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 $\mu = 4 - 2^{-21}$

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

\[ \begin{align*}
\text{Absolute approximation error} & \\
\text{Required bound} & \\
\end{align*} \]

\[ \begin{align*}
E_{\text{approx}} & \leq \theta, \\
\text{with} \quad \theta = 3 \cdot 2^{-29} \approx 6 \cdot 10^{-9} \\
E_{\text{eval}} & < \eta, \\
\text{with} \quad \eta \approx 7.4 \cdot 10^{-9}
\]
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: $E_{\text{eval}} \geq \eta$

$s^* = 3.935581684112548828125$ 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

![Graph showing absolute approximation error](image)
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: $E_{\text{eval}} \geq \eta$

$s^* = 3.935581684112548828125$ 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 $\mu = 4 - 2^{-21}$

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

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

- Example of \textit{binary32} implementation

  → launched on a 64 processor grid
  → 36127 subintervals found in several hours ($\approx 5h.$)
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 and perspectives
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>$x^{-1}$</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>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)
### Impact of dynamic target latency

<table>
<thead>
<tr>
<th></th>
<th>$x^{1/3}$</th>
<th>$x^{-1/3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degree $(n_x,n_y)$</td>
<td>(8,1)</td>
<td>(9,1)</td>
</tr>
<tr>
<td>Delay on the operands</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Static <em>target</em> latency</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>Dynamic <em>target</em> latency</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Latency on unbounded parallelism and on ST231</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

Latency (# cycles) on unbounded parallelism and on ST231

Impact of dynamic target latency

<table>
<thead>
<tr>
<th>Degree ((n_x, n_y))</th>
<th>(x^{1/3})</th>
<th>(x^{-1/3})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n_x, n_y)</td>
<td>(8, 1)</td>
<td>(9, 1)</td>
</tr>
<tr>
<td>Delay on the operand (s) ((#) cycles)</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Static target latency</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>Dynamic target latency</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Latency on unbounded parallelism and on ST231</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

Latency \((\#\) cycles\) on unbounded parallelism and on ST231

\[\Rightarrow\] Conclude on the **optimality in terms of polynomial evaluation latency**
## Numerical results

### Timings for code generation

<table>
<thead>
<tr>
<th></th>
<th>$x^{1/2}$</th>
<th>$x^{-1/2}$</th>
<th>$x^{1/3}$</th>
<th>$x^{-1/3}$</th>
<th>$x^{-1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degree ($n_x, n_y$)</td>
<td>(8,1)</td>
<td>(9,1)</td>
<td>(8,1)</td>
<td>(9,1)</td>
<td>(10,0)</td>
</tr>
<tr>
<td>Static target latency</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>Dynamic target latency</td>
<td>13</td>
<td>13</td>
<td>16</td>
<td>16</td>
<td>13</td>
</tr>
<tr>
<td>Latency on unbounded parallelism</td>
<td>13</td>
<td>13</td>
<td>16</td>
<td>16</td>
<td>13</td>
</tr>
<tr>
<td>Latency on ST231</td>
<td>13</td>
<td>14</td>
<td>16</td>
<td>16</td>
<td>13</td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Parenthesization generation</td>
<td>172ms</td>
<td>152ms</td>
<td>53s</td>
<td>56s</td>
<td>168ms</td>
</tr>
<tr>
<td>Arithmetic Operator Choice</td>
<td>6ms</td>
<td>6ms</td>
<td>7ms</td>
<td>11ms</td>
<td>4ms</td>
</tr>
<tr>
<td>Scheduling</td>
<td>29s</td>
<td>4m21s</td>
<td>32ms</td>
<td>132ms</td>
<td>7s</td>
</tr>
<tr>
<td>Certification (Gappa)</td>
<td>6s</td>
<td>4s</td>
<td>1m38s</td>
<td>1m07s</td>
<td>11s</td>
</tr>
<tr>
<td>Total time ($\approx$)</td>
<td>35s</td>
<td>4m25s</td>
<td>2m31s</td>
<td>2m03s</td>
<td>18s</td>
</tr>
</tbody>
</table>

Timing of each step of the generation flow
### Timings for code generation

<table>
<thead>
<tr>
<th></th>
<th>$x^{1/2}$</th>
<th>$x^{-1/2}$</th>
<th>$x^{1/3}$</th>
<th>$x^{-1/3}$</th>
<th>$x^{-1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degree $(n_x, n_y)$</td>
<td>(8,1)</td>
<td>(9,1)</td>
<td>(8,1)</td>
<td>(9,1)</td>
<td>(10,0)</td>
</tr>
<tr>
<td>Static target latency</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>Dynamic target latency</td>
<td>13</td>
<td>13</td>
<td>16</td>
<td>16</td>
<td>13</td>
</tr>
<tr>
<td>Latency on unbounded parallelism</td>
<td>13</td>
<td>13</td>
<td>16</td>
<td>16</td>
<td>13</td>
</tr>
<tr>
<td>Latency on ST231</td>
<td>13</td>
<td>14</td>
<td>16</td>
<td>16</td>
<td>13</td>
</tr>
<tr>
<td>Parenthesization generation</td>
<td>172ms</td>
<td>152ms</td>
<td>53s</td>
<td>56s</td>
<td>168ms</td>
</tr>
<tr>
<td>Arithmetic Operator Choice</td>
<td>6ms</td>
<td>6ms</td>
<td>7ms</td>
<td>11ms</td>
<td>4ms</td>
</tr>
<tr>
<td>Scheduling</td>
<td>29s</td>
<td>4m21s</td>
<td>32ms</td>
<td>132ms</td>
<td>7s</td>
</tr>
<tr>
<td>Certification (Gappa)</td>
<td>6s</td>
<td>4s</td>
<td>1m38s</td>
<td>1m07s</td>
<td>11s</td>
</tr>
<tr>
<td>Total time ($\approx$)</td>
<td>35s</td>
<td>4m25s</td>
<td>2m31s</td>
<td>2m03s</td>
<td>18s</td>
</tr>
</tbody>
</table>

**Timing of each step of the generation flow**

- Impact of the target latency on the first step of the generation
### Numerical results

#### Timings for code generation

<table>
<thead>
<tr>
<th></th>
<th>$x^{1/2}$</th>
<th>$x^{-1/2}$</th>
<th>$x^{1/3}$</th>
<th>$x^{-1/3}$</th>
<th>$x^{-1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degree ($n_x,n_y$)</td>
<td>(8,1)</td>
<td>(9,1)</td>
<td>(8,1)</td>
<td>(9,1)</td>
<td>(10,0)</td>
</tr>
<tr>
<td>Static target latency</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>Dynamic target latency</td>
<td>13</td>
<td>13</td>
<td>16</td>
<td>16</td>
<td>13</td>
</tr>
<tr>
<td>Latency on unbounded parallelism</td>
<td>13</td>
<td>13</td>
<td>16</td>
<td>16</td>
<td>13</td>
</tr>
<tr>
<td>Latency on ST231</td>
<td>13</td>
<td>14</td>
<td>16</td>
<td>16</td>
<td>13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Parenthesization generation</th>
<th>Arithmetic Operator Choice</th>
<th>Scheduling</th>
<th>Certification (Gappa)</th>
<th>Total time ($\approx$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>172ms 152ms 53s 56s 168ms</td>
<td>6ms 6ms 7ms 11ms 4ms</td>
<td>29s 4m21s 32ms 132ms 7s</td>
<td>6s 4s 1m38s 1m07s 11s</td>
<td>35s 4m25s 2m31s 2m03s 18s</td>
</tr>
</tbody>
</table>

**Timing of each step of the generation flow**

- Impact of the target latency on the first step of the generation
- **What may dominate the cost**
  - scheduling algorithm
  - certification using Gappa
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 and perspectives
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

⇒ 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
Perspectives

- Faithful implementation of floating-point operators
  - other floating-point operators:
    - \( \log_2(1 + x) \) over \([0.5, 1)\), \(\frac{1}{\sqrt{1 + x^2}}\) over \([0, 0.5)\), ...
  - roots and their reciprocals: rounding condition decision not automated yet
Faithful implementation of floating-point operators

- other floating-point operators:
  - \( \log_2(1 + x) \) over \([0.5, 1)\), \( 1/\sqrt{1 + x^2} \) over \([0, 0.5)\), ...

- roots and their reciprocals: rounding condition decision not automated yet

Extension to other binary floating-point formats

- square root in \( \text{binary}64 \): 171 cycles on ST231, 396 cycles with STlib
Perspectives

- Faithful implementation of floating-point operators
  - other floating-point operators:
    - $\log_2(1 + x)$ over $[0.5, 1)$, $1/\sqrt{1 + x^2}$ over $[0, 0.5)$, ...
  - roots and their reciprocals: rounding condition decision not automated yet

- Extension to other binary floating-point formats
  - square root in $\text{binary64}$: 171 cycles on ST231, 396 cycles with STlib

- Extension to other architectures, typically FPGAs
  - polynomial evaluation-based approach: already seems to be a good alternative to multiplicative methods on FPGAs
  - the other techniques introduced of this thesis: should be investigated further
Implementation of binary floating-point arithmetic on embedded integer processors

Polynomial evaluation-based algorithms and certified code generation

Guillaume Revy
Advisors: Claude-Pierre Jeannerod and Gilles Villard

Arénaire INRIA project-team (LIP, Ens Lyon)  Université de Lyon  CNRS