A new binary floating-point division algorithm and its implementation in software

Guillaume Revy

joint work with C.-P. Jeannerod, H. Knochel, C. Monat and G. Villard

Arénaire Inria Rhône-Alpes - project team
Laboratoire de l'Informatique du Parallélisme - ENS Lyon

Groupe de travail Arénaire (LIP - ENS Lyon)
Lyon - November 21, 2008
Context and objectives

Context

- FLIP library development
- software implementation of binary floating-point division
  → targets a VLIW integer processor of the ST200 family
- precision $p$, register size $k$, extremal exponents $(e_{\text{min}}, e_{\text{max}})$
  → $2 \leq p \leq e_{\text{max}}$ and $e_{\text{min}} = 1 - e_{\text{max}}$
- description of the algorithm in terms of the parameters $(k, p, e_{\text{max}})$
- implementation for the \textit{binary32 format} \( \Rightarrow (k, p, e_{\text{max}}) = (32, 24, 127) \)
- no support of \textit{subnormal} numbers
  → input/output: $\pm 0$, $\pm \infty$, qNaN, sNaN or \textit{normal} binary floating-point number

Objectives

- faster software implementation
- correct rounding-to-nearest-even ($\text{RN}_p$)
Outline of the talk

Properties and division algorithm

Sufficient conditions to ensure correct rounding

Generation and validation of efficient evaluation codes

Experimental results

Current work and conclusion
Outline of the talk

Properties and division algorithm

Sufficient conditions to ensure correct rounding

Generation and validation of efficient evaluation codes

Experimental results

Current work and conclusion
Floating-point data encoding

Definition
Let $x$ be a floating-point datum. Since subnormal numbers are not supported, $x$ is:

- either a special datum: $\pm 0$, $\pm \infty$, sNaN or qNaN,
- or a normal binary floating-point number

$$x = (-1)^{s_x} \cdot m_x \cdot 2^{e_x},$$

with $s_x \in \{0, 1\}$, $m_x = 1.m_{x,1} \ldots m_{x,p-1} \in [1, 2)$ and $e_x \in \{e_{\text{min}}, \ldots, e_{\text{max}}\}$. 
Floating-point data encoding

Definition
Let $x$ be a floating-point datum. Since subnormal numbers are not supported, $x$ is:

- either a special datum: $\pm 0$, $\pm \infty$, sNaN or qNaN,
- or a normal binary floating-point number

\[
x = (-1)^{s_x} \cdot m_x \cdot 2^{e_x},
\]
with $s_x \in \{0, 1\}$, $m_x = 1.m_{x,1} \ldots m_{x,p-1} \in [1, 2)$ and $e_x \in \{e_{\text{min}}, \ldots, e_{\text{max}}\}$.

Binary interchange encoding
Let $X$ be the $k$-bit unsigned integer encoding of $x$: $X = \sum_{i=0}^{k-1} X_i \cdot 2^i$.

\[
\begin{array}{ccc}
  s_x & E_x = e_x + e_{\text{max}} & m_x - 1 = 0.m_{x,1} \ldots m_{x,p-1} \\
  1 \text{ bit} & k - p \text{ bits} & p - 1 \text{ bits}
\end{array}
\]

$\Rightarrow E_x = \sum_{i=0}^{w-1} X_{i+p-1} \cdot 2^i$ and $X_i = m_{x,p-1-i}$ for $i = 0, \ldots, p - 1$. 
IEEE 754 specification

Let \( x, y \) be two binary floating-point data:

\[
x/y = (-1)^{s_r} \cdot |x|/|y|,
\]

with \( s_r = s_x \text{ XOR } s_y \).

\[
\begin{array}{c|c|c|c|c}
|x|/|y| & |y| & \hline \\
 & +0 & normal & +\infty & NaN \\
+0 & qNaN & +0 & +0 & qNaN \\
normal & +\infty & |x|/|y| & +0 & qNaN \\
+\infty & +\infty & +\infty & qNaN & qNaN \\
NaN & qNaN & qNaN & NaN & NaN \\
\end{array}
\]

Special values for \( |x|/|y| \).
IEEE 754 specification

Let $x, y$ be two binary floating-point data:

$$x/y = (-1)^{s_r} \cdot |x|/|y|,$$

with $s_r = s_x \ XOR \ s_y$.

| $x/|y|$ | $|y|$ |
|-------|-------|
| +0    | normal | $+\infty$ | NaN |
| normal| qNaN   | +0          | qNaN |
| $+\infty$ | $+\infty$ | qNaN | qNaN |
| NaN   | qNaN   | qNaN       | qNaN |

| $|x|$ | $|y|$ |
|-------|-------|
| +0    | qNaN   |
| $+\infty$ | qNaN |
| qNaN  | qNaN   |

Special values for $\text{RN}_p(|x|/|y|)$.

⇒ since $\text{RN}_p(-r) = -\text{RN}_p(r)$, for non special inputs:

$$\text{RN}_p(x/y) = (-1)^{s_r} \cdot \text{RN}_p(|x|/|y|).$$
Efficient special input handling

Let $X$ and $Y$ the unsigned integers encoding $|x|$ and $|y|$. How to detect if $|x|$ or $|y|$ is a special input?

Solution 1 $X == 0$ or $X \geq 2^{k-1} - 2^{p-1}$

<table>
<thead>
<tr>
<th>Value or range of integer $X$</th>
<th>Floating-point datum $x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>$+0$</td>
</tr>
<tr>
<td>$[2^{p-1}, 2^{k-1} - 2^{p-1})$</td>
<td>positive normal number</td>
</tr>
<tr>
<td>$2^{k-1} - 2^{p-1}$</td>
<td>$+\infty$</td>
</tr>
<tr>
<td>$(2^{k-1} - 2^{p-1}, 2^{k-1} - 2^{p-2})$</td>
<td>sNaN</td>
</tr>
<tr>
<td>$[2^{k-1} - 2^{p-2}, 2^{k-1})$</td>
<td>qNaN</td>
</tr>
</tbody>
</table>

Floating-point data encoded by $X$. 
Efficient special input handling

Let $X$ and $Y$ the unsigned integers encoding $|x|$ and $|y|$. How to detect if $|x|$ or $|y|$ is a special input?

**Solution 1** $X == 0$ or $X \geq 2^{k-1} - 2^{p-1}$

**Solution 2** integer addition modulo $2^k$ / 2’s complement representation

![Diagram](image)
Efficient special input handling

Let $X$ and $Y$ the unsigned integers encoding $|x|$ and $|y|$. How to detect if $|x|$ or $|y|$ is a special input?

**Solution 1**  
$X == 0$ or $X \geq 2^{k-1} - 2^{p-1}$

**Solution 2**  
integer addition modulo $2^k$ / 2’s complement representation

Guillaume Revy  
A new binary floating-point division algorithm and its implementation in software
Efficient special input handling

Let $X$ and $Y$ the unsigned integers encoding $|x|$ and $|y|$. How to detect if $|x|$ or $|y|$ is a special input?

**Solution 1** $X == 0$ or $X \geq 2^{k-1} - 2^{p-1}$

**Solution 2** integer addition modulo $2^k$ / 2’s complement representation

\[
\text{if } \max(X - 1, Y - 1) \geq 2^{k-1} - 2^{p-1} - 1
\]
Efficient special input handling

| \(|x|/|y|\)  | \(|y|\)  |
|---|---|
| +0  normal            +∞ NaN |
| +∞ Normal            +0 qNaN |
| +∞ +∞                qNaN qNaN |
| NaN qNaN            qNaN qNaN |

Special values for $\text{RN}_p(|x|/|y|)$.

Let $X$ and $Y$ the unsigned integers encoding $|x|$ and $|y|$.

$\Rightarrow$ if $\max(X - 1, Y - 1) \geq 2^{k-1} - 2^{p-1} - 1$

$\Rightarrow$ if $(X == Y \text{ OR } \max(X, Y) > 2^{k-1} - 2^{p-1}) \rightarrow qNaN$

$\Rightarrow$ if $(X < 2^{k-1} - 2^{p-1} \text{ AND } Y \neq 0) \rightarrow \pm 0$

$\Rightarrow$ else $\rightarrow \pm\infty$
General division algorithm

Let $x, y$ be two positive binary floating-point numbers. Then

$$x/y = m_x/m_y \times 2^{e_x-e_y},$$

that is, assuming $c = [m_x \geq m_y]$

$$x/y = (2m_x/m_y \cdot 2^{-c}) \times 2^{e_x-e_y-1+c},$$

with $\ell = (2m_x/m_y \cdot 2^{-c}) = \ell_0.\ell_1\ell_2 \ldots \ell_p\ell_{p+1} \ldots$ and $d = e_x - e_y - 1 + c$. 
General division algorithm

Let $x, y$ be two positive binary floating-point numbers. Then

$$x/y = m_x/m_y \times 2^{e_x-e_y},$$

that is, assuming $c = \lceil m_x \geq m_y \rceil$

$$x/y = (2m_x/m_y \cdot 2^{-c}) \times 2^{e_x-e_y-1+c},$$

with $\ell = (2m_x/m_y \cdot 2^{-c}) = \ell_0.\ell_1\ell_2 \ldots \ell_p\ell_{p+1}\ldots$ and $d = e_x - e_y - 1 + c$.

Property 1

*If* $m_x \geq m_y$ *then* $\ell \in [1, 2 - 2^{1-p}]$ *else* $\ell \in (1, 2 - 2^{1-p})$.

$$x/y = \ell \times 2^d \implies \text{RN}_p(x/y) = \text{RN}_p(\ell) \times 2^d,$$

*Remark:* the computation of the result exponent $d$ is trivial.
Underflow / Overflow detection

Since $\text{RN}_p(\ell) \in [1, 2 - 2^{1-p}] \Rightarrow$ no result exponent update is required

- **Overflow**: if $d \geq e_{\text{max}} + 1 \rightarrow +\infty$
- **Underflow**: if $d \leq e_{\text{min}} - 1 \rightarrow +0$
Underflow / Overflow detection

Since $\text{RN}_p(\ell) \in [1, 2 - 2^{1-p}] \Rightarrow$ no result exponent update is required

- **Overflow:** if $d \geq e_{\text{max}} + 1 \rightarrow +\infty$
- **Underflow:** if $d \leq e_{\text{min}} - 1 \rightarrow +0$

$\Rightarrow$ **exception:** if $(1 - 2^{-p}) \cdot 2^{e_{\text{min}}} \leq x/y < 2^{e_{\text{min}}}$

- “as if subnormals were supported” $\Rightarrow \text{RN}_p(x/y) = 2^{e_{\text{min}}}$
Underflow / Overflow detection

Since $\text{RN}_p (\ell) \in [1, 2 - 2^{1-p}] \Rightarrow$ no result exponent update is required

- **Overflow:** if $d \geq e_{\text{max}} + 1 \rightarrow +\infty$
- **Underflow:** if $d \leq e_{\text{min}} - 1 \rightarrow +0$

$\Rightarrow$ exception: if $(1 - 2^{-p}) \cdot 2^{e_{\text{min}}} \leq x/y < 2^{e_{\text{min}}}$
  - “as if subnormals were supported” $\rightarrow \text{RN}_p (x/y) = 2^{e_{\text{min}}}$

**Property 2**

*One has* $(1 - 2^{-p}) \cdot 2^{e_{\text{min}}} \leq x/y < 2^{e_{\text{min}}}$ *if and only if* $d = e_{\text{min}} - 1$ *and*

$m_x = 2 - 2^{1-p}$ *and* $m_y = 1$.

$\Rightarrow$ early detection
How to compute a correctly rounded significand?


Let \( v \) be a value that approximates \( \ell \) from above, such that

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

with \( v = 01.v_1v_2\ldots v_{k-2} \).
How to compute a correctly rounded significand?


Let $v$ be a value that approximates $\ell$ from above, such that

$$|(\ell + 2^{-p-1}) - v| < 2^{-p-1},$$

with $v = 01.v_1v_2 \ldots v_{k-2}$.

$$\Rightarrow w = v \text{ truncated after } p \text{ bits}$$

$$w = 01.v_1v_2 \ldots v_p00 \ldots 00 \quad \text{and} \quad -2^{-p} < \ell - w < 2^{-p}.$$
How to compute a correctly rounded significand?


Let $v$ be a value that approximates $\ell$ from above, such that

$$|(\ell + 2^{-p-1}) - v| < 2^{-p-1},$$

with $v = 01.v_1v_2 \ldots v_{k-2}$.

$\Rightarrow w = v$ truncated after $p$ bits

$$w = 01.v_1v_2 \ldots v_p00 \ldots 00$$

and

$$-2^{-p} < \ell - w < 2^{-p}.$$

**Property 3**

The value $\ell = 2m_x/m_y \cdot 2^{-c}$ cannot be halfway between two normal binary floating-point numbers.

$\Rightarrow$ implementation of the test $w \geq \ell$: $w \times m_y \geq 2m_x \cdot 2^{-c}$
Outline of the talk

Properties and division algorithm

Sufficient conditions to ensure correct rounding

Generation and validation of efficient evaluation codes

Experimental results

Current work and conclusion
General principle


Goal

Computation of the value $v$ such that $|(\ell + 2^{-p-1}) - v| < 2^{-p-1}$.

$\Rightarrow \ell + 2^{-p-1} =$ exact result of $F : (s, t) \mapsto 2^{-p-1} + s/(1 + t)$ at the point

$$(s^*, t^*) = (2m_x \cdot 2^{-c}, m_y - 1),$$

with $s^* \in S = [1, 2 - 2^{1-p}] \cup [2, 4 - 2^{3-p}]$ and $t^* \in T = [0, 1 - 2^{1-p}]$. 
General principle

Goal
Computation of the value \( v \) such that \(|(\ell + 2^{-p-1}) - v| < 2^{-p-1}\).

\[ \Rightarrow \quad \ell + 2^{-p-1} = \text{exact result of } F : (s, t) \mapsto 2^{-p-1} + s/(1 + t) \text{ at the point } (s^*, t^*) = (2m_x \cdot 2^{-c}, m_y - 1), \]

with \( s^* \in S = [1, 2 - 2^{1-p}] \cup [2, 4 - 2^{3-p}] \) and \( t^* \in T = [0, 1 - 2^{1-p}] \).

\[ \Rightarrow \quad \text{approximation of } F \text{ by a suitable bivariate polynomial } P \text{ over } S \times T : \]

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

▶ evaluation at run-time: smallest degree for polynomial \( a \)
General principle


Goal

Computation of the value \( v \) such that \(|(\ell + 2^{-p-1}) - v| < 2^{-p-1}|\.

\[ \Rightarrow \ell + 2^{-p-1} = \text{exact result of } F : (s, t) \mapsto 2^{-p-1} + s/(1 + t) \text{ at the point } (s^*, t^*) = (2m_x \cdot 2^{-c}, m_y - 1), \]

with \( s^* \in S = [1, 2 - 2^{1-p}] \cup [2, 4 - 2^{3-p}] \) and \( t^* \in T = [0, 1 - 2^{1-p}]\).

\[ \Rightarrow \text{approximation of } F \text{ by a suitable bivariate polynomial } P \text{ over } S \times T: \]

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

▶ evaluation at run-time: smallest degree for polynomial \( a \)

\[ \Rightarrow \text{evaluate } P \text{ with an accurately enough evaluation program } \mathcal{P} \]

▶ \( v = \mathcal{P}(s^*, t^*) \)
Approximation and rounding error conditions

Let $\alpha(a)$ and $\rho(P)$ be the approximation and rounding errors:

$$\alpha(a) = \max_{t \in T} |1/(1 + t) - a(t)| \quad \text{and} \quad \rho(P) = \max_{(s,t) \in S \times T} |P(s,t) - P(s,t)|.$$

We can check that

$$|(\ell + 2^{-p-1}) - v| \leq (4 - 2^{3-p})\alpha(a) + \rho(P).$$
Approximation and rounding error conditions


Let $\alpha(a)$ and $\rho(P)$ be the approximation and rounding errors:

$$
\alpha(a) = \max_{t \in T} |1/(1 + t) - a(t)| \quad \text{and} \quad \rho(P) = \max_{(s,t) \in S \times T} |P(s,t) - \mathcal{P}(s,t)|.
$$

We can check that

$$
|(\ell + 2^{-p-1}) - v| \leq (4 - 2^{3-p}) \alpha(a) + \rho(P).
$$

**Property 4**

If $(4 - 2^{3-p}) \alpha(a) + \rho(P) < 2^{-p-1}$ then $|(\ell + 2^{-p-1}) - v| < 2^{-p-1}$. 
Approximation and rounding error conditions


Let $\alpha(a)$ and $\rho(\mathcal{P})$ be the approximation and rounding errors:

$$
\alpha(a) = \max_{t \in \mathcal{T}} \left| \frac{1}{1 + t} - a(t) \right|
$$

and

$$
\rho(\mathcal{P}) = \max_{(s, t) \in \mathcal{S} \times \mathcal{T}} \left| P(s, t) - \mathcal{P}(s, t) \right|
$$

We can check that

$$
|\ell + 2^{-p-1} - v| \leq (4 - 2^{3-p})\alpha(a) + \rho(\mathcal{P})
$$

**Property 4**

If $(4 - 2^{3-p})\alpha(a) + \rho(\mathcal{P}) < 2^{-p-1}$ then $|\ell + 2^{-p-1} - v| < 2^{-p-1}$.

Since $\rho(\mathcal{P}) > 0$, the approximation error $\alpha(a)$ must satisfy

$$(4 - 2^{3-p})\alpha(a) < 2^{-p-1} \quad \text{i.e.} \quad \alpha(a) < 2^{-p-1} / (4 - 2^{3-p}).$$

Finally, the rounding error $\rho(\mathcal{P})$ must satisfy

$$
\rho(\mathcal{P}) < 2^{-p-1} - (4 - 2^{3-p})\alpha(a).
$$
Example for the binary32 implementation

Example

- polynomial degree $\delta = 10$
- truncated Remez’ polynomial / 32-bit coefficients
- $\alpha(a) \leq \theta_0 = 3 \cdot 2^{-29} \approx 2^{-27.41}$
- $\rho(P) < \eta_0 = 2^{-25} - (4 - 2^{-21}) \cdot \theta_0 \approx 2^{-26.9999} \rightarrow$ checked with Gappa?
Example for the binary32 implementation

Example

- polynomial degree $\delta = 10$
- truncated Remez’ polynomial / 32-bit coefficients
- $\alpha(a) \leq \theta_0 = 3 \cdot 2^{-29} \approx 2^{-27.41}$
- $\rho(P) < \eta_0 = 2^{-25} - (4 - 2^{-21}) \cdot \theta_0 \approx 2^{-26.9999} \rightarrow$ checked with $Gappa$?

$\Rightarrow$ the condition is not satisfied, particularly when $m_x < m_y$

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

$\rightarrow \rho(P) = 2^{-26.9988}$
Subdomain-based error conditions

⇒ splitting $\mathcal{T}$ into $n$ subintervals: $\mathcal{T} = \bigcup_{i=1}^{n} \mathcal{T}^{(i)}$

⇒ check that, for each subinterval $\mathcal{T}^{(i)}$,

$$(4 - 2^{3-p}) \cdot \alpha^{(i)}(a) + \rho^{(i)}(P) < 2^{-p-1}.$$
Implementation steps

1. determine minimal degree $\delta$ for polynomial $a$
2. compute a polynomial $a$ that satisfies $\alpha(a) < 2^{-p-1}/(4 - 2^{3-p})$
3. find in an automatic way an efficient evaluation code $P$
4. validate automatically the resulting evaluation program $P$
Outline of the talk

Properties and division algorithm

Sufficient conditions to ensure correct rounding

Generation and validation of efficient evaluation codes

Experimental results

Current work and conclusion
Description of the problem

Goal
Produce/validate automatically an efficient evaluation program $P$.

- target features:
  - 4 issues and at most 2 mul./cycle
  - latencies: addition = 1 cycle / multiplication = 3 cycles

- Horner’s scheme: $(3 + 1) \times 11 = 44$ cycles
  - sequential scheme
  - no ILP exposure

⇒ efficient = reduction of the evaluation latency / nb. of multiplications
⇒ express more ILP
Description of the problem

Data implementation

- fixed-point evaluation program: $V = \text{div\_eval}(S, T)$, with
  $$s^* = S \cdot 2^{-30}, \quad t^* = T \cdot 2^{-32} \quad \text{and} \quad v = V \cdot 2^{-30}$$
  with $S$ and $T$ computed from inputs $X$ and $Y$ respectively.

- implementation of polynomial coefficients in absolute value
  $$a(t) = \sum_{i=0}^{10} a_i t^i \quad \text{with} \quad a_i = (-1) \cdot A_i \cdot 2^{-32} \in (-1, 1).$$

  ⇒ the sign is not stored → appropriate choice of arithmetic operators

- implementation using only positive intermediate variables
Evaluation tree generation


**First step**: generate a set of efficient evaluation trees

- **Requirement / Assumption**:
  - operator cost: mul. = 3 cycles / add. = 1 cycle
  - delay between $S$ and $T$
  - unbounded parallelism
Evaluation tree generation


First step: generate a set of efficient evaluation trees

- Requirement / Assumption:
  - operator cost: mul. = 3 cycles / add. = 1 cycle
  - delay between $S$ and $T$
  - unbounded parallelism

- Two substeps:
  1. determine a target latency $\tau$
  2. generate automatically a set of evaluation trees, with height $\leq \tau$
Evaluation tree generation


First step: generate a set of efficient evaluation trees

- Requirement / Assumption:
  - operator cost: mul. = 3 cycles / add. = 1 cycle
  - delay between $S$ and $T$
  - unbounded parallelism

- Two substeps:
  1. determine a target latency $\tau$
  2. generate automatically a set of evaluation trees, with height $\leq \tau$

\[ \Rightarrow \text{number of evaluation trees = extremely large } \rightarrow \text{several filters} \]
\[ \Rightarrow \text{if no tree satisfies } \tau \text{ then increase } \tau \text{ and restart} \]
Example for the binary32 implementation

- Multiplication (3 cycles)
- Addition (1 cycle)

Diagram:

- Nodes labeled with variables (e.g., $r_0, r_1, r_2, \ldots$)
- Edges represent operations or dependencies
- Red nodes and arrows indicate critical path
- 14 cycles to reach the final result

Note: The diagram includes operations such as addition ($A_0, A_1, A_2, \ldots$) and multiplication ($0x20$), with specific cycles marked for clarity.
Arithmetic operator choice

**Second step**: handle coefficient signs through an appropriate arithmetic operator choice

- label evaluation tree by appropriate arithmetic operator: + or −
- polynomial coefficients are implemented in absolute value
- for example, $a_0 > 0$ and $a_1 < 0$
  \[ a_0 - |a_1|t \]  instead of  \[ a_0 + a_1 t \]
- ensure that all intermediate values have constant sign
Arithmetic operator choice

Second step: handle coefficient signs through an appropriate arithmetic operator choice

- label evaluation tree by appropriate arithmetic operator: + or −
- polynomial coefficients are implemented in absolute value
- for example, $a_0 > 0$ and $a_1 < 0$
  \[
  \Rightarrow a_0 - |a_1|t \text{ instead of } a_0 + a_1 t
  \]

- ensure that all intermediate values have constant sign
  \[
  \Rightarrow \text{if the sign of an intermediate value changes when the input varies then the evaluation tree is rejected}
  \]

⇒ implementation with MPFI
Example for the binary32 implementation

- multiplication (3 cycles)
- addition (1 cycle)
- subtraction (1 cycle)
Third step: check the practical scheduling

- schedule the evaluation tree on a simplified model of a real target architecture (operator costs / nb. issues / constraints on operators)
- check if no increase of latency
Scheduling verification


Third step: check the practical scheduling

- schedule the evaluation tree on a simplified model of a real target architecture (operator costs / nb. issues / constraints on operators)
- check if no increase of latency

⇒ if practical latency > theoretical latency then the evaluation tree is rejected

⇒ implementation using a naive list scheduling algorithm
### Example for the binary32 implementation

<table>
<thead>
<tr>
<th>Cycle</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>(r_0)</td>
<td>(r_4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(r_6)</td>
<td>(r_{13})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(r_{11})</td>
<td>(r_{20})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(r_1)</td>
<td>(r_5)</td>
<td>(r_{22})</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(r_2)</td>
<td>(r_{14})</td>
<td>(r_{19})</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(r_{12})</td>
<td>(r_{15})</td>
<td>(r_{21})</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>(r_7)</td>
<td>(r_{10})</td>
<td>(r_{23})</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>(r_3)</td>
<td>(r_8)</td>
<td>(r_{24})</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>(r_{16})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>(r_{17})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>(r_9)</td>
<td>(r_{25})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>(r_{18})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>(V)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Feasible scheduling on ST231.

⇒ 3 issues are enough
Objective
Find a splitting of $\mathcal{T}$ into $n$ subinterval(s) $\mathcal{T}^{(i)}$, and check that

$$(4 - 2^{3-p}) \cdot \alpha^{(i)}(a) + \rho^{(i)}(\mathcal{P}) < 2^{-p-1} \text{ for } i \in \{1, \ldots, n\}.$$ 

- implementation of the splitting by dichotomy

- for each $\mathcal{T}^{(i)}$
  1. compute an approximation error bound $\alpha^{(i)}$ with Sollya
  2. determine an evaluation error bound for $\rho^{(i)}(\mathcal{P})$
  3. check this bound with Gappa
  $\Rightarrow$ if this bound is not satisfied, $\mathcal{T}^{(i)}$ is split up into 2 subintervals

- launched on the LIP “grid”
- $\approx 5$ hours / 36127 subintervals found
Evaluation program validation strategy

* Does the condition

\[(4 - 2^{3-p}) \cdot \alpha^{(i)}(a) + \rho^{(i)}(P) < 2^{-p-1}\]

hold for \(i \in \{1, \ldots, n\}\)?

<table>
<thead>
<tr>
<th>Depth</th>
<th>Subintervals</th>
<th>(\alpha^{(\cdot)}(a) \leq)</th>
<th>(\rho^{(\cdot)}(P) &lt;)</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(l_{1,1} = [2^{-23}, 1 - 2^{-23}])</td>
<td>(\theta_1 \approx 2^{-27.41})</td>
<td>(\eta_1 \approx 2^{-26.99})</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>(l_{2,1} = [2^{-23}, 0.5 - 2^{-23}]) (l_{2,2} = [0.5, 1 - 2^{-23}])</td>
<td>(\theta_2 \approx 2^{-27.41}) (\theta_1 \approx 2^{-27.41})</td>
<td>(\eta_2 \approx 2^{-26.99})</td>
<td>yes</td>
</tr>
<tr>
<td>(\ldots)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(j)</td>
<td>(l_{j,1} = [2^{-23}, 0.5 - 2^{-23}]) (l_{j,2} = [0.5, 0.75 - 2^{-23}]) (l_{j,19309} = [0.921875, 0.92578113079071044921875]) (l_{j,19533} = [0.97490406036376953125, 0.97490441799163818359375])</td>
<td>(\theta_2 \approx 2^{-27.41}) (\theta_1 \approx 2^{-27.41}) (\theta_3 \approx 2^{-27.44}) (\theta_4 \approx 2^{-27.49})</td>
<td>(\eta_2 \approx 2^{-26.99}) (\eta_1 \approx 2^{-26.99}) (\eta_3 \approx 2^{-26.90}) (\eta_4 \approx 2^{-26.77})</td>
<td>yes</td>
</tr>
</tbody>
</table>

Splitting steps when \(m_x < m_y\).
Outline of the talk

Properties and division algorithm

Sufficient conditions to ensure correct rounding

Generation and validation of efficient evaluation codes

Experimental results

Current work and conclusion
Validation and performance evaluation

- Validation of the complete code:
  - the *Extremal Rounding Tests Set* (D.W. Matula)
  - *TestFloat* package
  - exhaustive tests on mantissa (with fixed result exponent)

- Performances evaluation on ST231 architecture
  - VLIW integer processor of ST200 family
Experimental results

Performances on ST231

<table>
<thead>
<tr>
<th>Nb. of instructions</th>
<th>Latency</th>
<th>IPC</th>
<th>Code size</th>
</tr>
</thead>
<tbody>
<tr>
<td>87</td>
<td>27 cycles</td>
<td>$87/27 \approx 3.22$</td>
<td>424 bytes</td>
</tr>
</tbody>
</table>

- if-conversion mechanism: fully straight-line assembly (branch-free)
- high IPC value: confirms the parallel nature of our approach
- 87 instructions: latency $\geq 1 (\text{slct/return}) + \lceil 85 \text{ instr.}/4 \text{ issues} \rceil = 23$
- speed-up by a factor of $\approx 1.78$ compared to the previous implementation (48 cycles)
Outline of the talk

Properties and division algorithm

Sufficient conditions to ensure correct rounding

Generation and validation of efficient evaluation codes

Experimental results

Current work and conclusion
Implementation of subnormal numbers support

- the exact result $x/y$ can be halfway between two consecutive subnormal binary floating-point numbers
  - the implementation of rounding test ($w \geq \ell$) is more complicated

- no need to detect underflow \textit{a priori}
  - directly detect through the rounding algorithm

- same principle / same polynomial evaluation
Future work and conclusion

- implementation of other rounding modes, with and without subnormal numbers support

- algorithmics of exception handling (inexact, division by zero, ...)
  - full IEEE 754-2008 compliance
  - what is the overhead?

- development of a binary floating-point division generator (already exists for square root)
  - automatic generation of division in other formats
  - validation of our approach

- acceleration of the validation of the resulting evaluation code