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

Objectives

- faster software implementation
- correct rounding-to-nearest-even (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: ± 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 $s_x \in \{0, 1\}$, $m_x = 1.m_{x,1} \dots m_{x,p-1} \in [1, 2)$ and $e_x \in \{e_{\mathrm{min}}, \dots, e_{\mathrm{max}}\}$.

Floating-point data encoding

Definition

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

- either a special datum: ± 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: $X = \sum_{i=0}^{k-1} X_i \cdot 2^i$.

s_x	$E_x = e_x + e_{\mathbf{max}}$	$m_x - 1 = 0.m_{x,1} \ldots m_{x,p-1}$
1 **bit**	$k - p$ **bits**	$p - 1$ **bits**

$\Rightarrow$ $E_x = \sum_{i=0}^{w-1} X_{i+p-1} \cdot 2^i$ 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 $s_r = s_x$ XOR s_y.

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

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
$\|x\|$	$+0$	qNaN	$+0$	$+0$	qNaN
	normal	$+\infty$	$\mathrm{RN}_p(\|x\|/\|y\|)$	$+0$	qNaN
	$+\infty$	$+\infty$	$+\infty$	qNaN	qNaN
	NaN	qNaN	qNaN	qNaN	qNaN

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

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

$$\mathrm{RN}_p(x/y) = (-1)^{s_r} \cdot \mathrm{RN}_p(|x|/|y|).$$

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 $X \geq 2^{k-1} - 2^{p-1}$

Value or range of integer X	Floating-point datum x
0	$+0$
$[2^{p-1}, 2^{k-1} - 2^{p-1})$	positive normal number
$2^{k-1} - 2^{p-1}$	$+\infty$
$(2^{k-1} - 2^{p-1}, 2^{k-1} - 2^{p-2})$	sNaN
$[2^{k-1} - 2^{p-2}, 2^{k-1})$	qNaN

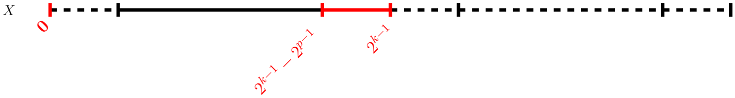
Floating-point data encoded by X.

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

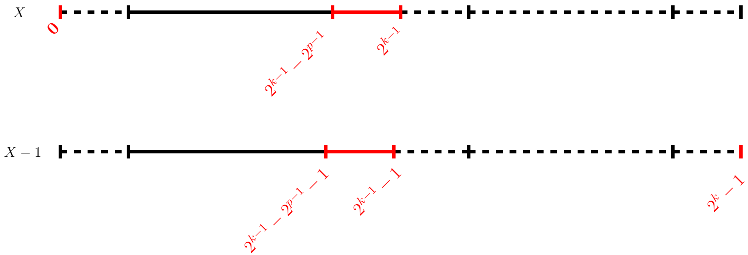


Efficient special input handling

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

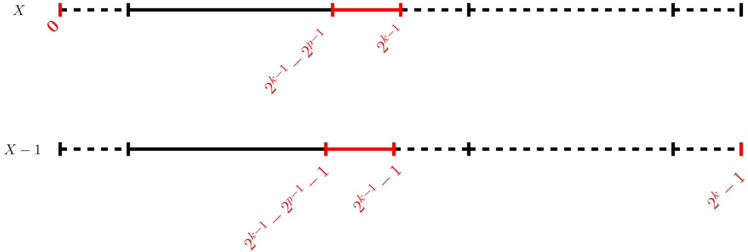


Efficient special input handling

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



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

Efficient special input handling

$\lvert x\rvert / \lvert y\rvert$		$\lvert y\rvert$			
		$+0$	normal	$+\infty$	NaN
$\lvert x\rvert$	$+0$	qNaN	$+0$	$+0$	qNaN
	normal	$+\infty$	RN$_p(\lvert x\rvert/\lvert y\rvert)$	$+0$	qNaN
	$+\infty$	$+\infty$	$+\infty$	qNaN	qNaN
	NaN	qNaN	qNaN	qNaN	qNaN

Special values for RN$_p(\lvert x\rvert/\lvert y\rvert)$.

Let X and Y the unsigned integers encoding $\lvert x\rvert$ and $\lvert y\rvert$.

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

- if $\left(X == Y \text{ OR } \max(X,Y) > 2^{k-1} - 2^{p-1}\right) \rightarrow$ qNaN
- if $\left(X < 2^{k-1} - 2^{p-1} \text{ AND } Y \neq 0\right) \rightarrow \pm 0$
- 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 $c = [m_x \geq m_y]$

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

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 $c = [m_x \geq m_y]$

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

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* $\ell \in (1, 2 - 2^{1-p})$*.*

$$x/y = \ell \times 2^d \;\Rightarrow\; \mathrm{RN}_p\left(x/y\right) = \mathrm{RN}_p(\ell) \times 2^d, \text{ with } \quad \mathrm{RN}_p\left(\ell\right) \in [1, 2 - 2^{1-p}].$$

Remark: the computation of the result exponent d is trivial.

Underflow / Overflow detection

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

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

Underflow / Overflow detection

Since $\mathrm{RN}_p(\ell) \in [1, 2 - 2^{1-p}] \Rightarrow$ 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" $\rightarrow \mathrm{RN}_p(x/y) = 2^{e_{\text{min}}}$

Underflow / Overflow detection

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

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

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

- "as if subnormals were supported" $\rightarrow \mathrm{RN}_p(x/y) = 2^{e_{\mathrm{min}}}$

Property 2

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

$\Rightarrow$ early detection

How to compute a correctly rounded significand ?

M.D. Ercegovac & T. Lang, *Digital Arithmetic*, 2004.

Let v be a value that approximates ℓ from above, such that

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

with $v = 01.v_1 v_2 \ldots v_{k-2}$.

How to compute a correctly rounded significand ?

M.D. Ercegovac & T. Lang, *Digital Arithmetic*, 2004.

Let v be a value that approximates ℓ from above, such that

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

with $v = 01.v_1 v_2 \ldots v_{k-2}$.

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

$$w = 01.v_1 v_2 \ldots v_p 00 \ldots 00 \qquad \text{and} \qquad -2^{-p} < \ell - w < 2^{-p}.$$

How to compute a correctly rounded significand ?

M.D. Ercegovac & T. Lang, *Digital Arithmetic*, 2004.

Let v be a value that approximates ℓ from above, such that

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

with $v = 01.v_1 v_2 \dots v_{k-2}$.

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

$$w = 01.v_1 v_2 \dots v_p 00 \dots 00 \qquad \text{and} \qquad -2^{-p} < \ell - w < 2^{-p}.$$

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

C.P. Jeannerod, H. Knochel, C. Monat & G. Revy, *Computing floating-point square roots via bivariate polynomial evaluation*, 2008.

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

General principle

C.P. Jeannerod, H. Knochel, C. Monat & G. Revy, *Computing floating-point square roots via bivariate polynomial evaluation*, 2008.

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

$\Rightarrow$ approximation of F by a suitable bivariate polynomial P over $\mathcal{S} \times \mathcal{T}$:

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

- evaluation at run-time: smallest degree for polynomial a

General principle

C.P. Jeannerod, H. Knochel, C. Monat & G. Revy, *Computing floating-point square roots via bivariate polynomial evaluation*, 2008.

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

$\Rightarrow$ approximation of F by a suitable bivariate polynomial P over $\mathcal{S} \times \mathcal{T}$:

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

- evaluation at run-time: smallest degree for polynomial a

$\Rightarrow$ evaluate P with an accurately enough evaluation program $\mathcal{P}$

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

Approximation and rounding error conditions

C.P. Jeannerod, H. Knochel, C. Monat & G. Revy, *Computing floating-point square roots via bivariate polynomial evaluation*, 2008.

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

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

We can check that

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

Approximation and rounding error conditions

C.P. Jeannerod, H. Knochel, C. Monat & G. Revy, *Computing floating-point square roots via bivariate polynomial evaluation*, 2008.

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

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

We can check that

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

Property 4

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

Approximation and rounding error conditions

C.P. Jeannerod, H. Knochel, C. Monat & G. Revy, *Computing floating-point square roots via bivariate polynomial evaluation*, 2008.

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

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

We can check that

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

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

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

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
- $\alpha(a) \leq \theta_0 = 3 \cdot 2^{-29} \approx 2^{-27.41}$
- $\rho(\mathcal{P}) < \eta_0 = 2^{-25} - (4 - 2^{-21}) \cdot \theta_0 \approx 2^{-26.9999}$ → checked with *Gappa* ?

Example for the binary32 implementation

Example

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

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

$s^* = 3.935581684112548828125$ and $t^* = 0.97490441799163818359375$

$$\rightarrow \quad \rho(\mathcal{P}) = 2^{-26.9988}$$

Subdomain-based error conditions

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

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

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

Implementation steps

1. determine minimal degree δ 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 $\mathcal{P}$
4. validate automatically the resulting evaluation program $\mathcal{P}$

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 $\mathcal{P}$.

- target features:
  - $\longrightarrow$ 4 issues and at most 2 mul./cycle
  - $\longrightarrow$ latencies: addition = 1 cycle / multiplication = 3 cycles
- Horner's scheme: $(3+1) \times 11 = 44$ cycles
  - $\longrightarrow$ sequential scheme
  - $\longrightarrow$ no ILP exposure

$\Rightarrow$ *efficient* = reduction of the evaluation latency / nb. of multiplications

$\Rightarrow$ express more ILP

Description of the problem

Data implementation

- fixed-point evaluation program: $V = div\_eval(S, T)$, 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

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

  $\Rightarrow$ the sign is not stored $\rightarrow$ appropriate choice of arithmetic operators

- implementation using only positive intermediate variables

Evaluation tree generation

J. Harrison, T. Kubaska, S. Story & P.T.P. Tang, *The computation of transcendental functions on IA-64 architecture*, 1999.

First step: generate a set of efficient evaluation trees

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

Evaluation tree generation

J. Harrison, T. Kubaska, S. Story & P.T.P. Tang, *The computation of transcendental functions on IA-64 architecture*, 1999.

First step: generate a set of efficient evaluation trees

- Requirement / Assumption:
  - $\longrightarrow$ operator cost: mul. = 3 cycles / add. = 1 cycle
  - $\longrightarrow$ delay between S and T
  - $\longrightarrow$ unbounded parallelism
- Two substeps:
  1. determine a target latency τ
  2. generate automatically a set of evaluation trees, with height $\leq \tau$

Evaluation tree generation

J. Harrison, T. Kubaska, S. Story & P.T.P. Tang, *The computation of transcendental functions on IA-64 architecture*, 1999.

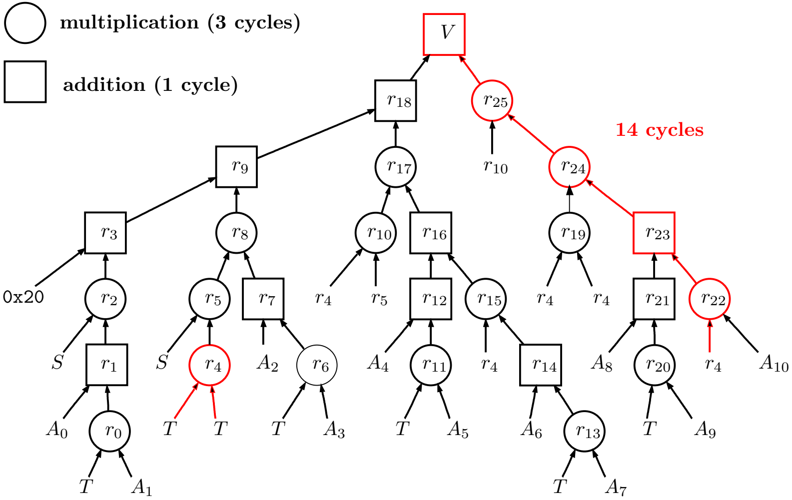
First step: generate a set of efficient evaluation trees

- Requirement / Assumption:
  - $\longrightarrow$ operator cost: mul. = 3 cycles / add. = 1 cycle
  - $\longrightarrow$ delay between S and T
  - $\longrightarrow$ unbounded parallelism
- Two substeps:
  1. determine a target latency τ
  2. generate automatically a set of evaluation trees, with height $\leq \tau$

$\Rightarrow$ number of evaluation trees = extremely large $\longrightarrow$ several filters

$\Rightarrow$ if no tree satisfies τ then increase τ and restart

Example for the binary32 implementation



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 $a_1 < 0$

$$\Rightarrow \qquad a_0 - |a_1|t \quad \text{instead of} \quad a_0 + a_1 t$$

- 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 $a_1 < 0$

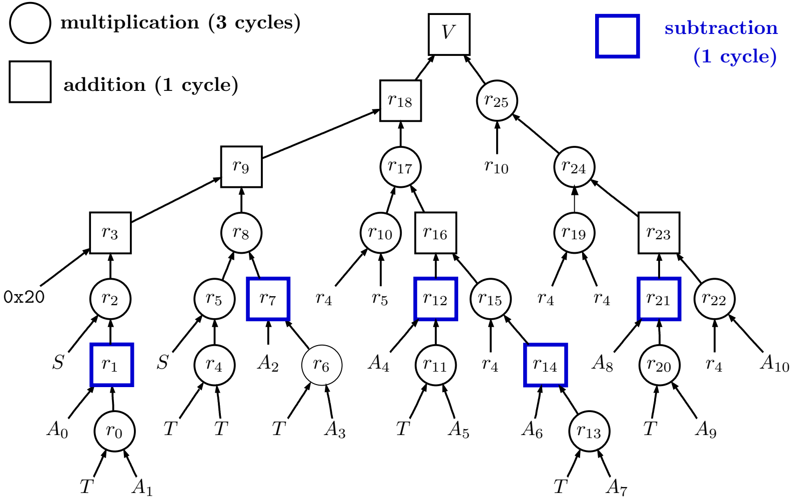
$$\Rightarrow \qquad a_0 - |a_1|t \quad \text{instead of} \quad a_0 + a_1 t$$

- ensure that all intermediate values have constant sign

$\Rightarrow$ if the sign of an intermediate value changes when the input varies then the evaluation tree is rejected

$\Rightarrow$ implementation with MPFI

Example for the binary32 implementation



Scheduling verification

J. Harrison, T. Kubaska, S. Story & P.T.P. Tang, *The computation of transcendental functions on IA-64 architecture*, 1999.

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

J. Harrison, T. Kubaska, S. Story & P.T.P. Tang, *The computation of transcendental functions on IA-64 architecture*, 1999.

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

$\Rightarrow$ if practical latency $>$ theoretical latency then the evaluation tree is rejected

$\Rightarrow$ implementation using a naive list scheduling algorithm

Example for the binary32 implementation

	Issue 1	Issue 2	Issue 3	Issue 4
Cycle 0	r_0	r_4		
Cycle 1	r_6	r_{13}		
Cycle 2	r_{11}	r_{20}		
Cycle 3	r_1	r_5	r_{22}	
Cycle 4	r_2	r_{14}	r_{19}	
Cycle 5	r_{12}	r_{15}	r_{21}	
Cycle 6	r_7	r_{10}	r_{23}	
Cycle 7	r_3	r_8	r_{24}	
Cycle 8	r_{16}			
Cycle 9	r_{17}			
Cycle 10	r_9	r_{25}		
Cycle 11				
Cycle 12	r_{18}			
Cycle 13	V			

Feasible scheduling on ST231.

$\Rightarrow$ 3 issues are enough

Demo

Evaluation program validation strategy

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 $\rho^{(\mathcal{P})}$
  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”
- ≈ 5 hours / 36127 subintervals found

Evaluation program validation strategy

* Does the condition

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

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

Depth	Subintervals	$\alpha^{(\cdot)}(a) \leq$	$\rho^{(\cdot)}(\mathcal{P}) <$	*
1	$I_{1,1} = [2^{-23}, 1 - 2^{-23}]$	$\theta_1 \approx 2^{-27.41}$	$\eta_1 \approx 2^{-26.99}$	no
2	$I_{2,1} = [2^{-23}, 0.5 - 2^{-23}]$	$\theta_2 \approx 2^{-27.41}$	$\eta_2 \approx 2^{-26.99}$	yes
	$I_{2,2} = [0.5, 1 - 2^{-23}]$	$\theta_1 \approx 2^{-27.41}$	$\eta_1 \approx 2^{-26.99}$	no
...				
j	$I_{j,1} = [2^{-23}, 0.5 - 2^{-23}]$	$\theta_2 \approx 2^{-27.41}$	$\eta_2 \approx 2^{-26.99}$	yes
	$I_{j,2} = [0.5, 0.75 - 2^{-23}]$	$\theta_1 \approx 2^{-27.41}$	$\eta_1 \approx 2^{-26.99}$	yes
	$I_{j,19309} = [0.921875, 0.92578113079071044921875]$	$\theta_3 \approx 2^{-27.44}$	$\eta_3 \approx 2^{-26.90}$	yes
	$I_{j,19533} = [0.97490406036376953125, 0.97490441799163818359375]$	$\theta_4 \approx 2^{-27.49}$	$\eta_4 \approx 2^{-26.77}$	yes

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:
  - $\longrightarrow$ the *Extremal Rounding Tests Set* (D.W. Matula)
  - $\longrightarrow$ *TestFloat* package
  - $\longrightarrow$ exhaustive tests on mantissa (with fixed result exponent)
- Performances evaluation on ST231 architecture
  - $\longrightarrow$ VLIW integer processor of ST200 family

Performances on ST231

Nb. of instructions	Latency	IPC	Code size
87	27 cycles	$87/27 \approx 3.22$	424 bytes

- if-conversion mechanism: fully straight-line assembly (branch-free)
- high IPC value: confirms the parallel nature of our approach
- 87 instructions: latency ≥ 1 ($\texttt{slct/return}$) + $\lceil 85$ instr.$/4$ issues$\rceil = 23$
- speed-up by a factor of ≈ 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
  - $\longrightarrow$ the implementation of rounding test ($w \geq \ell$) is more complicated
- no need to detect underflow *a priori*
  - $\longrightarrow$ 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,...)
  - $\longrightarrow$ full IEEE 754-2008 compliance
  - $\longrightarrow$ what is the overhead ?
- development of a binary floating-point division generator (already exists for square root)
  - $\longrightarrow$ automatic generation of division in other formats
  - $\longrightarrow$ validation of our approach
- acceleration of the validation of the resulting evaluation code