Séminaire – Équipe DALI (Laboratoire ELIAUS – Université de Perpignan)
Perpignan, France, June 23, 2009

Fast and accurate floating-point division on ST231

Algorithm, implementation, and automatic generation and validation

Guillaume Revy
Advisors: Claude-Pierre Jeannerod and Gilles Villard

Arénaire Inria project-team (LIP, ENS Lyon)   Université de Lyon   CNRS

Context and objectives

Context

- FLIP software library
  - $\longrightarrow$ `http://flip.gforge.inria.fr/`
  - $\longrightarrow$ support for floating-point arithmetic on integer processors
- low latency implementation of binary floating-point division
  - $\longrightarrow$ targets a VLIW integer processor of the ST200 family
- no support of *subnormal* numbers
  - $\longrightarrow$ input/output: ± 0, $\pm\infty$, NaN or *normal* number

Objectives

- faster software implementation (compared to FLIP 0.3)
  - $\longrightarrow$ expose instruction-level parallelism via bivariate polynomial evaluation
- correctly rounded
  - $\longrightarrow$ rounding-to-nearest even

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

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

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

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

Notation and assumptions

- Input (x, y): two positive normal numbers

  $\longrightarrow$ precision p, extremal exponents $(e_{\text{min}}, e_{\text{max}})$

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

- Computation: k-bit unsigned integers

  $\longrightarrow$ register size k

- Example for binary32 format: $(k, p, e_{\text{max}}) = (32, 24, 127)$

Outline of the talk

Division via polynomial evaluation

Generation of an efficient evaluation program

Validation of the generated evaluation program

Experimental results

Concluding remarks

Outline of the talk

Division algorithm flowchart

- Definition

$$c = \begin{cases} 1 & \text{if } m_x \geq m_y, \\ 0 & \text{if } m_x < m_y. \end{cases}$$

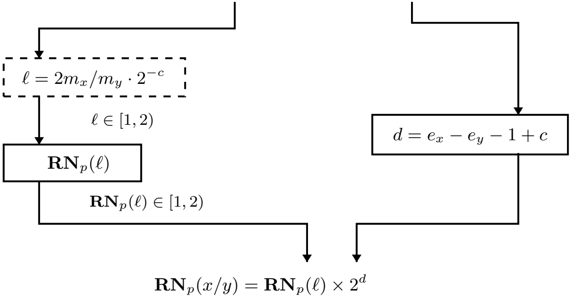
Division algorithm flowchart

- Definition

$$c = \begin{cases} 1 & \text{if } m_x \geq m_y, \\ 0 & \text{if } m_x < m_y. \end{cases}$$

- Range reduction

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



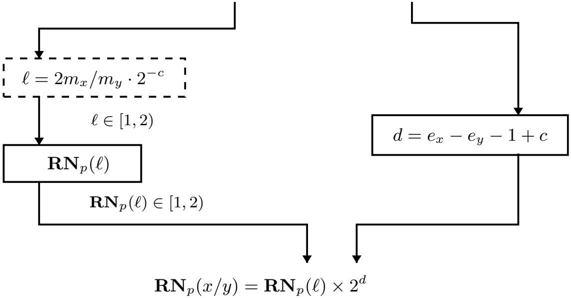
Division algorithm flowchart

- Definition

$$c = \begin{cases} 1 & \text{if } m_x \geq m_y, \\ 0 & \text{if } m_x < m_y. \end{cases}$$

- Range reduction

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



How to compute the correctly rounded significand $\mathrm{RN}_p(\ell)$?

How to compute a correctly rounded significand ?

- Iterative methods (restoring, non-restoring, ...)
  - Oberman and Flynn (1997)
  - minimal instruction-level parallelism exposure, sequential algorithm

How to compute a correctly rounded significand ?

- Iterative methods (restoring, non-restoring, ...)
  - Oberman and Flynn (1997)
  - minimal instruction-level parallelism exposure, sequential algorithm
- Multiplicative methods (Newton-Raphson, Goldschmidt)
  - Piñeiro and Bruguera (2002) – Raina's Ph.D/FLIP (2006)
  - more instruction-level parallelism exposure
  - previous implementation of division (FLIP 0.3)

How to compute a correctly rounded significand ?

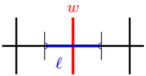
- Iterative methods (restoring, non-restoring, ...)
  - Oberman and Flynn (1997)
  - minimal instruction-level parallelism exposure, sequential algorithm
- Multiplicative methods (Newton-Raphson, Goldschmidt)
  - Piñeiro and Bruguera (2002) – Raina's Ph.D/FLIP (2006)
  - more instruction-level parallelism exposure
  - previous implementation of division (FLIP 0.3)
- Polynomial-based methods
  - Agarwal, Gustavson and Schmookler (1999)
    $\longrightarrow$ univariate polynomial evaluation
  - Our approach
    $\longrightarrow$ single bivariate polynomial evaluation

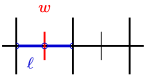
Truncated one-sided approximation

- See for example, Ercegovac and Lang (2004)
- 3 steps
  1. compute $v = (01.v_1 \dots v_{k-2})$ such that

     $$-2^{-p} \leq \ell - v < 0 \text{ that is implied by } |(\ell + 2^{-p-1}) - v| < 2^{-p-1}$$

  2. truncate v after p fraction bits: $w = (01.v_1 \dots v_p 0 \dots 0)$





  3. obtain $\mathrm{RN}_p(\ell)$ after possibly adding 2^{-p}

Truncated one-sided approximation

- See for example, Ercegovac and Lang (2004)
- 3 steps
  1. compute $v = (01.v_1 \dots v_{k-2})$ such that

     $$-2^{-p} \leq \ell - v < 0 \text{ that is implied by } |(\ell + 2^{-p-1}) - v| < 2^{-p-1}$$

  2. truncate v after p fraction bits: $w = (01.v_1 \dots v_p 0 \dots 0)$

  3. obtain $\mathrm{RN}_p(\ell)$ after possibly adding 2^{-p}

How to compute the one-sided approximation v ?

Computation of the one-sided approximation

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 $s^* = 2^{1-c} m_x$ and $t^* = m_y - 1$:

$$\ell + 2^{-p-1} = F(s^*, t^*).$$

Computation of the one-sided approximation

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 $s^* = 2^{1-c} m_x$ and $t^* = m_y - 1$:

$$\ell + 2^{-p-1} = F(s^*, t^*).$$

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

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

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

   $\longrightarrow$ approximation entails an error ϵ_{approx}

Computation of the one-sided approximation

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 $s^* = 2^{1-c}m_x$ and $t^* = m_y - 1$:
$$\ell + 2^{-p-1} = F(s^*, t^*).$$

2. Approximate $F(s,t)$ by a bivariate polynomial $P(s,t)$
$$P(s,t) = s \cdot a(t) + 2^{-p-1}.$$
   - $\longrightarrow$ $a(t)$: univariate polynomial approximant of $1/(1+t)$
   - $\longrightarrow$ approximation entails an error ϵ_{approx}

3. Evaluate $P(s,t)$ by a well-chosen efficient evaluation program $\mathcal{P}$
$$v = \mathcal{P}(s^*, t^*).$$
   - $\longrightarrow$ evaluation entails an error ϵ_{eval}

Computation of the one-sided approximation

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 $s^* = 2^{1-c} m_x$ and $t^* = m_y - 1$:

$$\ell + 2^{-p-1} = F(s^*, t^*).$$

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 entails an error ϵ_{approx}

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

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

$\rightarrow$ evaluation entails an error ϵ_{eval}

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

Sufficient error bounds

- Since by triangular inequality

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

with

$$\mu = \max\{s^*\} = \max\{2^{1-c} m_x\} = (4 - 2^{3-p})$$

Sufficient error bounds

- Since by triangular inequality

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

with

$$\mu = \max\{s^*\} = \max\{2^{1-c} m_x\} = (4 - 2^{3-p})$$

- One has to ensure

$$\mu \cdot \epsilon_{\text{approx}} + \epsilon_{\text{eval}} < 2^{-p-1}$$

- Sufficient conditions can be obtained

$$\epsilon_{\text{approx}} < 2^{-p-1}/\mu \quad \text{and} \quad \epsilon_{\text{eval}} < 2^{-p-1} - \mu \cdot \epsilon_{\text{approx}}$$

Implementation steps

1. determine the minimal degree δ for the polynomial approximant a
2. compute the polynomial approximant a such that
$$\epsilon_{\text{approx}} < 2^{-p-1}/\mu$$
3. find an efficient evaluation program $\mathcal{P}$ such that
$$\epsilon_{\text{eval}} < 2^{-p-1} - \mu \cdot \epsilon_{\text{approx}}$$
4. validate the evaluation program

$\Rightarrow$ implemented using *Sollya* (steps 1 and 2) and *Gappa* (step 4)

Outline of the talk

Automatic generation of an efficient evaluation program

- Evaluation program $\mathcal{P}$ = main part of the full software implementation
  $\longrightarrow$ dominates the cost

- By efficient, one means an evaluation program that
  - $\longrightarrow$ reduces the evaluation latency
  - $\longrightarrow$ reduces the number of multiplications
  - $\longrightarrow$ is accurate enough

Automatic generation of an efficient evaluation program

- Evaluation program $\mathcal{P}$ = main part of the full software implementation
  $\longrightarrow$ dominates the cost

- By efficient, one means an evaluation program that
  $\longrightarrow$ reduces the evaluation latency
  $\longrightarrow$ reduces the number of multiplications
  $\longrightarrow$ is accurate enough

- Target architecture : ST231
  $\longrightarrow$ 4-issue VLIW integer processor with at most 2 mul. per cycle
  $\longrightarrow$ latencies: addition = 1 cycle, multiplication = 3 cycles

Automatic generation of an efficient evaluation program

- Evaluation program $\mathcal{P}$ = main part of the full software implementation
  $\longrightarrow$ dominates the cost
- By efficient, one means an evaluation program that
  $\longrightarrow$ reduces the evaluation latency
  $\longrightarrow$ reduces the number of multiplications
  $\longrightarrow$ is accurate enough
- Target architecture : ST231
  $\longrightarrow$ 4-issue VLIW integer processor with at most 2 mul. per cycle
  $\longrightarrow$ latencies: addition = 1 cycle, multiplication = 3 cycles

Which evaluation program to evaluate the polynomial $P(s,t)$?

Example for the binary32 implementation: $(k, p) = (32, 24)$

$$P(s,t) = 2^{-p-1} + s \cdot \sum_{i=0}^{10} a_i t^i$$

- Horner's scheme: $(3+1) \times 11 = 44$ cycles
  - $\longrightarrow$ sequential scheme, no instruction-level parallelism exposure

Example for the binary32 implementation: $(k,p) = (32,24)$

$$P(s,t) = 2^{-p-1} + s \cdot \sum_{i=0}^{10} a_i t^i$$

- Horner's scheme: $(3+1) \times 11 = 44$ cycles
  - $\longrightarrow$ sequential scheme, no instruction-level parallelism exposure
- Estrin's scheme: 20 cycles
  - $\longrightarrow$ more instruction-level parallelism
  - $\longrightarrow$ a last multiplication by s
  - $\longrightarrow$ 2 cycles save by distributing the multiplication by s in the evaluation of the univariate polynomial $a(t)$

Example for the binary32 implementation: $(k, p) = (32, 24)$

$$P(s,t) = 2^{-p-1} + s \cdot \sum_{i=0}^{10} a_i t^i$$

- Horner's scheme: $(3+1) \times 11 = 44$ cycles
  - $\longrightarrow$ sequential scheme, no instruction-level parallelism exposure
- Estrin's scheme: 20 cycles
  - $\longrightarrow$ more instruction-level parallelism
  - $\longrightarrow$ a last multiplication by s
  - $\longrightarrow$ 2 cycles save by distributing the multiplication by s in the evaluation of the univariate polynomial $a(t)$
- ...

We can do much better.

- But how to explore the solution space and choose an efficient evaluation program ?
  - $\longrightarrow$ interest of automatic generation

Efficient evaluation tree generation

- Similar to Harrison, Kubaska, Story and Tang (1999)
- Assumption
  - $\longrightarrow$ unbounded parallelism
  - $\longrightarrow$ latencies of arithmetic operators: $+$ and $\times$

Efficient evaluation tree generation

- Similar to Harrison, Kubaska, Story and Tang (1999)
- Assumption
  - $\longrightarrow$ unbounded parallelism
  - $\longrightarrow$ latencies of arithmetic operators: $+$ and $\times$
- Two sub-steps
  1. determine a target latency τ

     $$\text{ie. } \tau = 3 \times \lceil \log_2(\deg(P)) \rceil + 1$$

  2. generate automatically a set of evaluation trees, with height $\leq \tau$

Efficient evaluation tree generation

- Similar to Harrison, Kubaska, Story and Tang (1999)
- Assumption
  - $\longrightarrow$ unbounded parallelism
  - $\longrightarrow$ latencies of arithmetic operators: $+$ and $\times$
- Two sub-steps
  1. determine a target latency τ

     ie. $\tau = 3 \times \lceil \log_2(\deg(P)) \rceil + 1$

  2. generate automatically a set of evaluation trees, with height $\leq \tau$

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

Efficient evaluation tree generation

- Similar to Harrison, Kubaska, Story and Tang (1999)
- Assumption
  - $\longrightarrow$ unbounded parallelism
  - $\longrightarrow$ latencies of arithmetic operators: $+$ and $\times$
- Two sub-steps
  1. determine a target latency τ

     ie. $\tau = 3 \times \lceil \log_2(\deg(P)) \rceil + 1$

  2. generate automatically a set of evaluation trees, with height $\leq \tau$

  $\Rightarrow$ if no tree satisfies τ then increase τ and restart
- Number of evaluation trees = extremely large $\longrightarrow$ several filters

Efficient evaluation tree generation

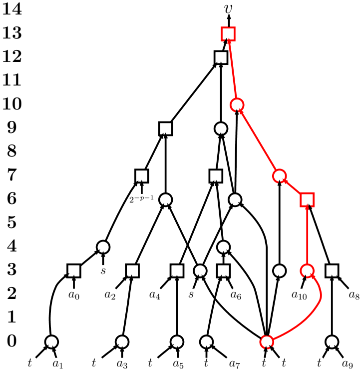
$$P(s,t) = 2^{-p-1} + s \cdot \sum_{i=0}^{10} a_i t^i$$

- first target latency $\tau = 13$
  $\longrightarrow$ no tree found

Efficient evaluation tree generation

$$P(s,t) = 2^{-p-1} + s \cdot \sum_{i=0}^{10} a_i t^i$$

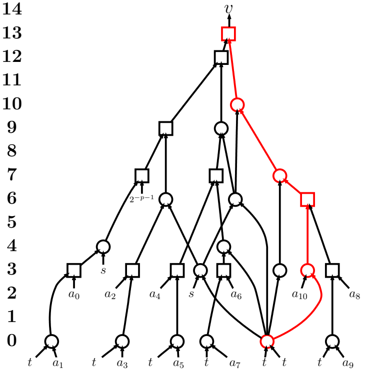
- first target latency $\tau = 13$
  $\longrightarrow$ no tree found
- second target latency $\tau = 14$
  $\longrightarrow$ obtained in about 10 sec.



Efficient evaluation tree generation

$$P(s,t) = 2^{-p-1} + s \cdot \sum_{i=0}^{10} a_i t^i$$

- first target latency $\tau = 13$
  $\longrightarrow$ no tree found
- second target latency $\tau = 14$
  $\longrightarrow$ obtained in about 10 sec.
- distribute the multiplication by s
  $\longrightarrow$ otherwise: 18 cycles
- too difficult to find such tree by hand



Arithmetic operator choice

- Polynomial coefficients implemented in absolute value
- All intermediate values have constant sign

  $\Rightarrow$ not store the sign: more accuracy

Arithmetic operator choice

- Polynomial coefficients implemented in absolute value
- All intermediate values have constant sign

  $\Rightarrow$ not store the sign: more accuracy

- Label evaluation trees by appropriate arithmetic operator: $+$ or $-$

Arithmetic operator choice

- Polynomial coefficients implemented in absolute value
- All intermediate values have constant sign

  $\Rightarrow$ not store the sign: more accuracy

- Label evaluation trees by appropriate arithmetic operator: $+$ or $-$
- If the sign of an intermediate value changes when the input varies then the evaluation tree is rejected

  $\Rightarrow$ implementation with certified interval arithmetic (MPFI)

Practical scheduling checking

- Schedule the evaluation trees on a simplified model of a real target architecture
  - $\longrightarrow$ operator costs, nb. issues, constraints on operators
  - $\longrightarrow$ no syllables constraint

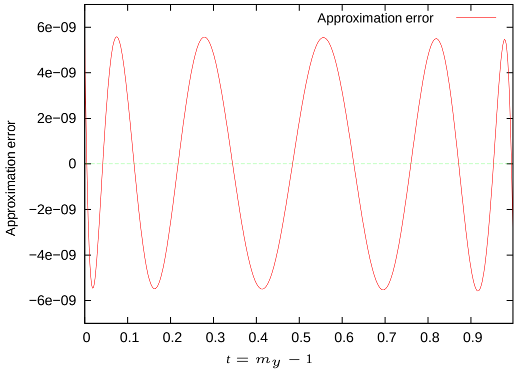
Practical scheduling checking

- Schedule the evaluation trees on a simplified model of a real target architecture
  - $\longrightarrow$ operator costs, nb. issues, constraints on operators
  - $\longrightarrow$ no syllables constraint
- Check if no increase of latency in practice compared to the latency on unbounded parallelism
  - $\Rightarrow$ if practical latency $>$ theoretical latency then the evaluation tree is rejected
  - $\Rightarrow$ implementation using naive list scheduling algorithm is enough

Outline of the talk

Example for the binary32 implementation: $(k, p) = (32, 24)$

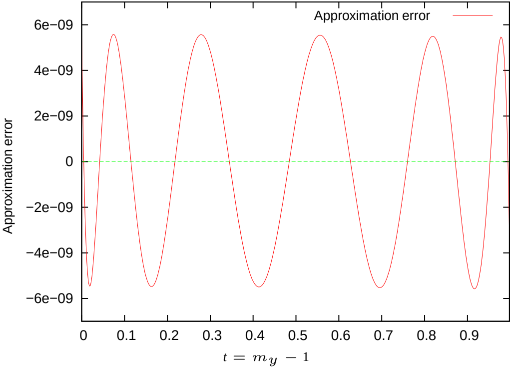
- Approximation of $1/(1+t)$ by truncated Remez' polynomial of degree 10



$$\epsilon_{\text{approx}} \leq 2^{-27.41\ldots} \approx 6.0 \text{ e-9} \quad < \quad 2^{-25}/(4 - 2^{-21}) \approx 7.4 \text{ e-9}$$

Example for the binary32 implementation: $(k, p) = (32, 24)$

- Approximation of $1/(1+t)$ by truncated Remez' polynomial of degree 10



$$\epsilon_{\text{approx}} \leq 2^{-27.41\ldots} \approx 6.0 \text{ e-9} \quad < \quad 2^{-25}/(4 - 2^{-21}) \approx 7.4 \text{ e-9}$$

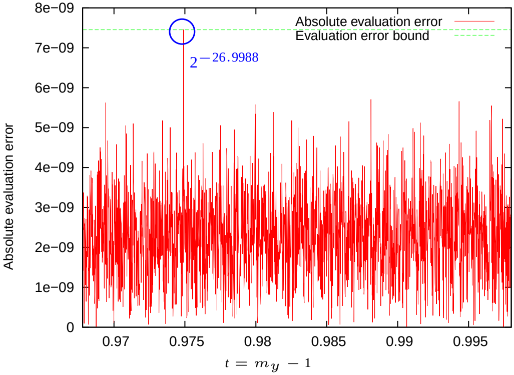
- Deduction of the evaluation error bound from ϵ_{approx}

$$\epsilon_{\text{eval}} < 2^{-25} - (4 - 2^{-21}) \cdot 2^{-27.41\ldots} \approx 2^{-26.9999\ldots} \approx 7.4 \text{ e-9}.$$

Example for the binary32 implementation: $(k, p) = (32, 24)$

- Case 1: $m_x \geq m_y \rightarrow$ condition satisfied
- Case 2: $m_x < m_y \rightarrow$ condition not satisfied

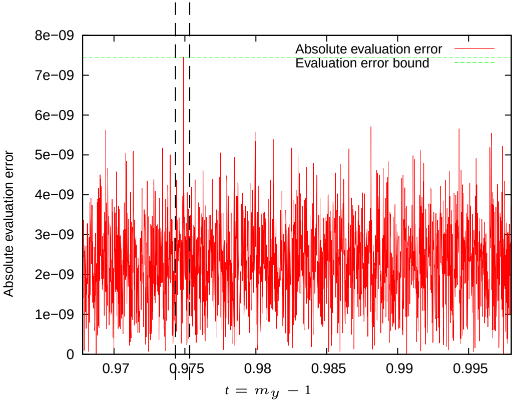
ie. $s^* = 3.935581684112548828125$ and $t^* = 0.974904417991638183594375$



Example for the binary32 implementation: $(k, p) = (32, 24)$

- Case 1: $m_x \geq m_y \rightarrow$ condition satisfied
- Case 2: $m_x < m_y \rightarrow$ condition not satisfied

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

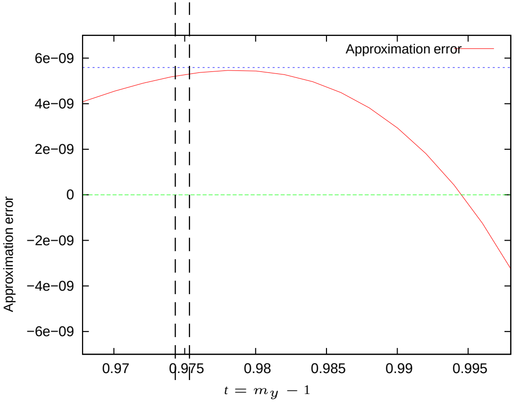


1. determine an interval $\mathcal{I}$ around this point

Example for the binary32 implementation: $(k,p) = (32,24)$

- Case 1: $m_x \geq m_y \rightarrow$ condition satisfied
- Case 2: $m_x < m_y \rightarrow$ condition not satisfied

ie. $s^* = 3.935581684112548828125$ and $t^* = 0.974904417991638183359375$



1. determine an interval $\mathcal{I}$ around this point
2. compute ϵ_{approx} over $\mathcal{I}$
3. determine an evaluation error bound η
4. check if $\epsilon_{\text{eval}} < \eta$?

Evaluation program validation strategy

- Find a splitting of the input interval into n subinterval(s) $\mathcal{T}^{(i)}$, and check that
$$\mu \cdot \epsilon_{\text{approx}}^{(i)} + \epsilon_{\text{eval}}^{(i)} < 2^{-p-1}$$
on each subinterval.

Evaluation program validation strategy

- Find a splitting of the input interval into n subinterval(s) $\mathcal{T}^{(i)}$, and check that
$$\mu \cdot \epsilon_{\text{approx}}^{(i)} + \epsilon_{\text{eval}}^{(i)} < 2^{-p-1}$$
on each subinterval.

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

  $\Rightarrow$ if this bound is not satisfied, $\mathcal{T}^{(i)}$ is split up into 2 subintervals
  - implemented using *Sollya* (steps 1 and 2) and *Gappa* (step 3)

Evaluation program validation strategy

- Find a splitting of the input interval into n subinterval(s) $\mathcal{T}^{(i)}$, and check that
$$\mu \cdot \epsilon_{\text{approx}}^{(i)} + \epsilon_{\text{eval}}^{(i)} < 2^{-p-1}$$
on each subinterval.

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

  $\Rightarrow$ if this bound is not satisfied, $\mathcal{T}^{(i)}$ is split up into 2 subintervals
  - implemented using *Sollya* (steps 1 and 2) and *Gappa* (step 3)

- Example of binary32 implementation

  $\longrightarrow$ launched on a 64 processor grid

  $\longrightarrow$ 36127 subintervals found in several hours (≈ 5h.)

Evaluation program validation strategy

* Does the condition

$$\mu \cdot \epsilon_{\text{approx}}^{(i)} + \epsilon_{\text{eval}}^{(i)} < 2^{-p-1}$$

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

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

Splitting steps when $m_x < m_y$.

Outline of the talk

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
  - → 4-issue VLIW integer processor of ST200 family
  - → at most 2 mul. per cycle
  - → latencies: addition = 1 cycle, multiplication = 3 cycles

Experimental results

Performances on ST231

	Nb. of instructions	Latency (# cycles)	IPC	Code size (bytes)
rounding to nearest	86	27	3.18	416

- speed-up by a factor of about 1.78 in rounding to nearest compared to the previous implementation (48 cycles)
  - optimized implementation
  - efficient ST200 compiler (`st200cc`)
- high IPC value: confirms the parallel nature of our approach

Outline of the talk

Concluding remarks

Contributions

- New approach for the implementation of binary floating-point division
  - $\longrightarrow$ bivariate polynomial-based algorithm
  - $\longrightarrow$ automatic generation and validation of efficient evaluation program
  - $\longrightarrow$ implementation targeted ST231 VLIW integer processor
- Speed-up by a factor of about 1.78 in rounding to nearest compared to the previous implementation

Since then

- Extension to subnormal numbers support
  - $\longrightarrow$ implementation in 31 cycles: 4 extra cycles
- Implementation of other functions

	Latency (# cycles)	IPC	Code size (bytes)	Speed-up
square root	21	2.47	276	2.38
reciprocal	22	2.59	336	1.75
reciprocal square root	29	2.24	368	2.27