Optimizing correctly-rounded reciprocal square roots for embedded VLIW cores

Claude-Pierre Jeannerod[1,2] and **Guillaume Revy**[2,1]

[1]INRIA Grenoble - Rhône-Alpes (Arénaire project-team, LIP, ENS Lyon) [2]Université de Lyon

1. Introduction

Context & Motivation

- Implementation of an efficient software support for IEEE 754 floating-point arithmetic on integer processors
  - set of correctly-rounded mathematical operators, handling of subnormal numbers, and handling of special inputs
  - development of the FLIP library [1] for the *binary32* floating-point format
- Optimized for the ST231 processor
  - 4-issue VLIW integer processor from the ST200 processor family (STMicroelectronics) → no FPU
  - integer processor for embedded media systems → highly used in audio and video domains (HD-IPTV, cell phones, wireless terminals, PDAs)

Purpose of our work

- Software implementation of correctly-rounded reciprocal square root ($x^{-1/2}$) → 29 cycles
  - frequently used in digital signal processing [6]
  - correctly-rounded implementation recommended by the latest revision of the IEEE 754 standard [2, §9.2]
- Optimized for the *binary32* format and the ST231 core
- Correctly-rounded RoundTiesToEven (rounding to nearest)
- Extension of our bivariate polynomial evaluation-based method introduced in [4] for square root
- Efficiency achieved by exploiting at best the instruction-level parallelism (ILP) of the ST231

Example of application

- Typical use of reciprocal square root = 3D vector normalization

$$[x, y, z] \mapsto [x/w, y/w, z/w]$$

with

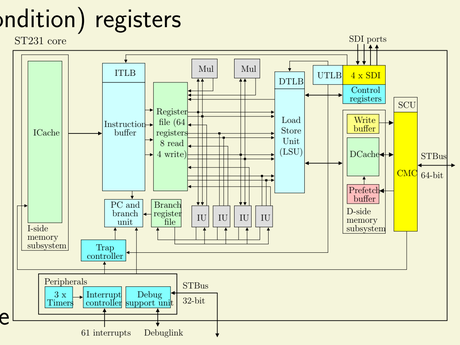
$$w = \sqrt{x^2 + y^2 + z^2}$$

- Without reciprocal square root:
  → 1 sqrt (23 cycles) + 3 div (3× 32 cycles) = 119 cycles
  With reciprocal square root:
  → 1 rsqrt (29 cycles) + 3 mul (3× 21 cycles) = 92 cycles
- Latency reduction by over 20 %

2. ST231 architecture and compiler

ST231, a 4-issue VLIW 32-bit embedded integer architecture

- 4 parallel ALU's / 2 parallel pipelined $32 \times 32 \to 32$-bit multipliers
- 1 leading zero counter
- Predicate execution → select instruction to remove branch penalty
- 64 general purpose 32-bit registers / 8 1-bit *branch* (condition) registers
- Efficient 32-bit immediate operand encoding



ST231 Compiler

- Open64 compiler technology
- Instruction level parallelism extractor and scheduler
- Select-based *if-conversion* → straightline assembly code
  → sequences of *select* instructions instead of costly control flow
- Linear Assembly Optimizer (LAO): generates schedule very close to the optimal

3. Some properties of reciprocal square root

Handling of special operands $x \in \{x < 0, \pm 0, \pm\infty, \text{NaN}\}$

- Filter out special operands using the standard binary interchange encoding format
- Compute special results required by [2] in parallel with the generic case

Positive finite operand x (precision $p \geq 2$)

Input: *binary32* floating-point number: $x = m \cdot 2^e = m' \cdot 2^{e'}$, with $m' = m \cdot 2^{\lambda}$, $e' = e - \lambda$, and

$$e' \in \mathbb{N} \cap [e_{\min} - p + 1, e_{\max}] \quad \text{and} \quad m' = 1.m_{\lambda+1} \cdots m_{p-1}$$

Output: $\text{RN}(x^{-1/2})$ = correct rounding-to-nearest of $x^{-1/2}$

$$x^{-1/2} = \ell \cdot 2^d \quad \text{and} \quad \text{RN}(x^{-1/2}) = \text{RN}(\ell) \cdot 2^d, \quad \text{and} \quad \ell \in [1, 2] \quad \text{and} \quad e_{\min} \leq d \leq e_{\max}$$

and where

$$d = -(e + 1 - c)/2, \quad \ell = s\sqrt{2/(1+t)}, \quad \text{and} \quad c = [e' \text{ is odd}]$$

Two useful properties

- $x^{-1/2}$ falls in the range of normal floating-point numbers
- $x^{-1/2}$ cannot be halfway between two consecutive floating-point numbers

4. How to approximate the exact value ℓ?

Existing method in FLIP 0.3: multiplicative method

- Initial approximation: degree-3 univariate polynomial
- Refinement by Goldschmidt's iteration [3]

Our approach: one-sided truncated approximation [4]

- Approximation of ℓ from above by v

$$2^{-p} < \ell - v \leq 0 \quad \text{and} \quad |\ell + 2^{-p-1} - v| < 2^{-p-1}$$

Computation of u = truncation of v after p fraction bits

$$0 \leq v - u < 2^{-p} \quad \text{and} \quad |\ell - u| < 2^{-p}$$

- Approximation v = result of the evaluation of a single bivariate polynomial

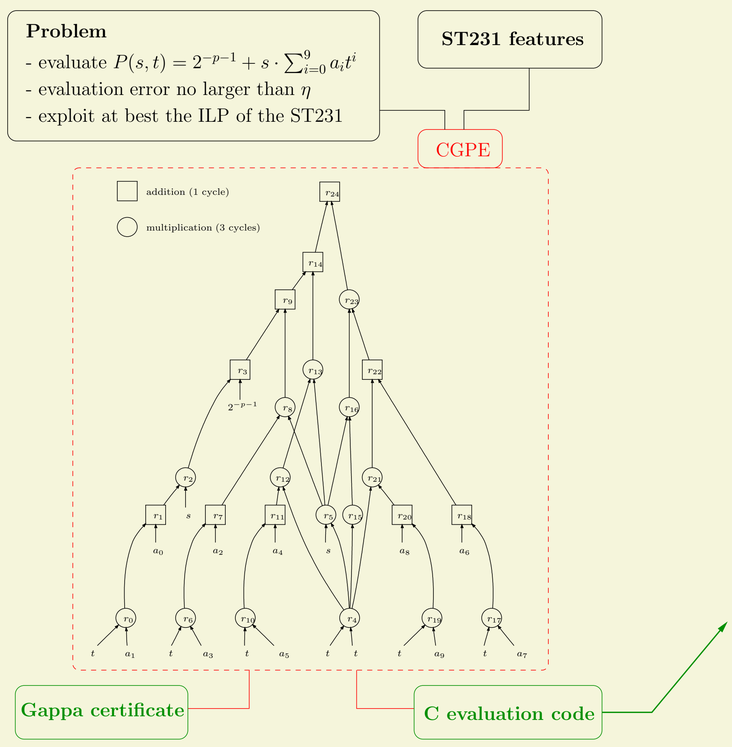
$$P(s, t) = 2^{-p-1} + s \cdot a(t)$$

with $a(t)$ a degree-9 truncated Remez approximant computed with Sollya

How to evaluate $P(s, t)$ efficiently?

- Horner's rule: 38 cycles, no ILP exposure
- Efficient and certified parenthesization automatically generated using CGPE [5]
  - Reduction of evaluation latency
    → 13 cycles on unbounded parallelism, 14 cycles on ST231
  - Evaluation error checked with Gappa
    → ensure correct rounding

CGPE generation flowchart



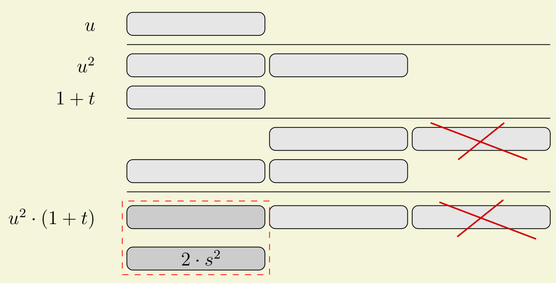

```
uint32_t __rsqrt_eval__(uint32_t T, uint32_t S)
{
  uint32_t r0 = mul(T, 0x5a82685d);          // 1.31
  uint32_t r1 = 0xb504f31f - r0;             // 1.31
  uint32_t r2 = mul(S, r1);                  // 2.30
  uint32_t r3 = 0x00000020 + r2;             // 2.30
  uint32_t r4 = mul(T, T);                   // 0.32
  uint32_t r5 = mul(S, r4);                  // 1.31
  uint32_t r6 = mul(T, 0x386fd5f4);          // 1.31
  uint32_t r7 = 0x43df72f7 - r6;             // 1.31
  uint32_t r8 = mul(r5, r7);                 // 2.30
  uint32_t r9 = r3 + r8;                     // 2.30
  uint32_t r10 = mul(T, 0x28724100);         // 1.31
  uint32_t r11 = 0x308b1798 - r10;           // 1.31
  uint32_t r12 = mul(r4, r11);               // 1.31
  uint32_t r13 = mul(r5, r12);               // 2.30
  uint32_t r14 = r9 + r13;                   // 2.30
  uint32_t r15 = mul(r4, r4);                // 0.32
  uint32_t r16 = mul(r5, r15);               // 1.31
  uint32_t r17 = mul(T, 0x106c5cd9);         // 1.31
  uint32_t r18 = 0x1d7bf968 - r17;           // 1.31
  uint32_t r19 = mul(T, 0x00fa9aa4);         // 1.31
  uint32_t r20 = 0x05dfffa4 - r19;           // 1.31
  uint32_t r21 = mul(r4, r20);               // 1.31
  uint32_t r22 = r18 + r21;                  // 1.31
  uint32_t r23 = mul(r16, r22);              // 2.30
  uint32_t r24 = r14 + r23;                  // 2.30
  return r24;
}
/* Feasible scheduling
 * > issues      : 4
 * > multipliers : 2
 ----------------------------------------------
 |          |  Is1  |  Is2  |  Is3  |  Is4  |
 ----------------------------------------------
 | Cycle  0 |   r0  |   r4  |       |       |
 | Cycle  1 |   r6  |  r10  |       |       |
 | Cycle  2 |  r17  |  r19  |       |       |
 | Cycle  3 |   r1  |   r5  |       |       |
 | Cycle  4 |   r2  |  r15  |       |       |
 | Cycle  5 |  r11  |  r20  |       |       |
 | Cycle  6 |  r12  |  r21  |       |       |
 | Cycle  7 |   r7  |  r16  |       |       |
 | Cycle  8 |   r8  |  r18  |       |       |
 | Cycle  9 |   r3  |  r13  |   r22 |       |
 | Cycle 10 |  r23  |       |       |       |
 | Cycle 11 |   r9  |       |       |       |
 | Cycle 12 |  r14  |       |       |       |
 | Cycle 13 |  r24  |       |       |       |
 ----------------------------------------------
 */
```

5. How to deduce RN(ℓ) from the approximation v?

- Deduce RN(ℓ): decide whether $u \geq \ell$, which is equivalent to

$$u^2 \cdot (1 + t) \geq 2 \cdot s^2, \quad \text{with} \quad 2 \cdot s^2 \in \{2, 4\}$$

- u, t, and $2 \cdot s^2$ exactly representable with 32 bits



- Computing the first 64 bits of the exact product are enough
- Test done on the first 32 bits of the exact product

6. Validation and performances

- Exhaustive comparison with Glibc and MPFR
- Performances on ST231

FLIP 1.0 (with subnormal)	FLIP 1.0 (without subnormal)	FLIP 0.3 (without subnormal)	Speedup
29 cycles	28 cycles	67 cycles	2.3

- Interest of the specialization of the reciprocal square root operator

Code sequence used for computing $x^{-1/2}$	Number N of instructions	Latency L (cycles)	N/L
`div(1.0f,sqrt(x))`	147 [124]	53 [47]	2.7 [2.6]
`inv(sqrt(x))`	121 [115]	49 [47]	2.5 [2.4]
`rsqrt(x)`	68 [63]	29 [28]	2.3 [2.2]

Some references

[1] FLIP - Floating-point Library for Intger Processors. Available at `http://flip.gforge.inria.fr/`.

[2] IEEE standard for floating-point arithmetic. IEEE Std. 754-2008, pp.1-58, Aug. 29 2008.

[3] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004.

[4] Claude-Pierre Jeannerod, Hervé Knochel, Christophe Monat, and Guillaume Revy. Computing floating-point square roots via bivariate polynomial evaluation. Technical Report RR2008-38, LIP, October 2008.

[5] Guillaume Revy. CGPE - Code Generation for Polynomial Evaluation. Available at `http://cgpe.gforge.inria.fr/`.

[6] Michael J. Schulte and Kent E. Wires. High-speed inverse square roots. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (ARITH-14)*, pages 124–131. IEEE Computer Society, 1999.

43rd Asilomar Conference on Signals, Systems and Computers – Pacific Grove, CA, USA, 1-4 November, 2009.