High-Radix Floating-Point Division Algorithms for Embedded VLIW Integer Processors *(Extended Abstract)*

Claude-Pierre Jeannerod, Saurabh-Kumar Raina and Arnaud Tisserand
Arénaire project (CNRS–ENS Lyon–INRIA–UCBL)
LIP, ENS Lyon. 46 allée d'Italie.
F–69364 LYON Cedex 07, FRANCE
Phone: +33 4 72 72 80 00, Fax: +33 72 72 80 80
E-mail: {firstame.lastname}@ens-lyon.fr

***Abstract*— This work presents floating-point division algorithms and implementations for embedded VLIW integer processors. On those processors, there is no hardware floating-point unit, for cost reasons. But, for portability and/or accuracy reasons, a software floating-point emulation layer is sometime useful. In this paper, we focus on high-radix digit-recurrence algorithms for floating-point division on integer VLIW processors. Our algorithms are targeted for the ST200 processor from STMicroelectronics.**

***Index Terms*— computer arithmetic, floating-point arithmetic, division, digit-recurrence algorithm, SRT algorithm, high-radix algorithm, integer processor, embedded processor, VLIW processor.**

I. Introduction

The implementation of fast floating-point (FP) division is not a trivial task [1], [2]. A study from Oberman and Flynn [3] shows that even if the number of issued division instructions is low (around 3% for SPEC benchmarks), the total duration of the division computation cannot be neglected (up to 40% of the time in arithmetic units).

General purpose processors allow fast FP division thanks to a dedicated unit based on the SRT (from the initials of Sweeney, Robertson, and Tocher) algorithm or Newton-Raphson's algorithm using the FMA (fused multiply and add) of the FP unit(s). Even when a software solution, such as Newton-Raphson's algorithm, is used, there is some minimal hardware support for those algorithms (e.g., seed tables, see [4]. The main division algorithms and implementations used in general purpose processors can be found in a complete survey [5].

Most special purpose or embedded processors for digital signal processing, image processing and digital control rely on *integer* or *fixed-point processors*, for cost reasons (small area). When implementing algorithms dealing with real numbers on such processors, one has to introduce some scaling operations in the target program, in order to keep accurate computations [6]. The insertion of scaling operations is complicated due to the wide range of real numbers required in many applications and it depends on the algorithm and data. Furthermore, scaling is time consuming at both the application and design levels. Of course, algorithms based on FP arithmetic do not have this problem. Then, porting programs from general purpose processors (with FP unit) to integer or fixed-point cores is a complex task. For instance, prototyped applications using Matlab have to be converted to fixed-point format.

Furthermore, circuits in these application fields seldom integrate dedicated division units. But in order to avoid slow software routines, manufacturers sometimes insert a division step instruction in the ALU (*arithmetic and logic unit*). Most of the time, this instruction is one step of the non-restoring division algorithm (radix-2). Then a complete n-bit division is done using n steps of this instruction. One goal of this work is to show that some other fast division algorithms may be well suited for the native integer (or fixed-point) hardware support of embedded processors.

This work is part of *Floating-Point Library for Integer Processor* (FLIP). This is a C library for the software support of single precision floating-point (FP) arithmetic on processors without FP hardware units such as VLIW or DSP processor cores for embedded applications. Our first target for FLIP is the ST200 family of VLIW processors from STMicroelectronics (see [7], [8]). Our algorithms can be easily extended to other processors with similar characteristics.

In this paper, we focus on pure software division implementation based on high-radix SRT algorithms. We investigate the implementations of these algorithms on processors with rectangular multipliers (e.g., $16 \times 32 \rightarrow 32$), more or less long latencies for the multiplication unit and parallel functional units (multipliers and ALU).

The paper is organized as follows. Section II gives definitions and notations, and recalls the basic algorithms used for software division. Section III presents the SRT algorithm and its extension to high-radix iterations. The implementation on the ST200 VLIW processor and its comparison to other solutions are presented in Section IV. In Section V, we briefly present the use of presented algorithms in the case of the FLIP library (*Floating-Point Library for Integer Processor*).

II. Basic Software Division

Digit-recurrence algorithms produce a fixed number of quotient bits at each iteration [9], [1]. Such algorithms are

similar to the "paper-and-pencil" method. The two basic digit-recurrence algorithms are the *restoring* and the *non-restoring* algorithms. Those algorithms only rely on very simple radix-2 iterations, i.e. one bit of the quotient is produced at each iteration. They are often used in software implementations for low performance applications.

Both Newton-Raphson's and Goldschmidt's algorithms are based on a functional iteration. The idea is to find an iteration that converges on the quotient value. As an example, Newton-Raphson's algorithm uses the following functional iteration to evaluate the reciprocal $1/d$:

$$y[j+1] = y[j] \times (2 - d \times y[j])$$

where $y[j]$ is the reciprocal approximation at iteration j.

The convergence of this solution is quadratic, that is, the number of correct bits doubles at every iteration. The obtained reciprocal should be multiplied by x in order to complete the computation of the quotient. The initial approximation y_0 can be looked up into a table to reduce the number of iterations. This kind of algorithm requires simple arithmetic operators and full-width multiplications (at least in the last iteration). In the following, we will not use this kind of solution because of its long latency for medium precision and because getting a correctly rounded result is much more complicated than using digit-recurrence algorithms.

A. Definitions and Notations

In this paper we follow the definitions and notations proposed in [1]. The division operation is defined by:

$$x = q \times d + rem$$

and

$$|rem| < |d| \times ulp \quad \text{and} \quad \text{sign}(rem) = \text{sign}(x)$$

where x is the *dividend*, d the *divisor*, q the *quotient*, and optionally rem the *remainder*. In our case, we have $0 \leq x < d$ and $d \in [1, 2)$. The *unit in the last place* is $ulp = r^{-n}$ for the radix-r representation of n-digit fractional quotients. In the following, we will use radix-2 or radix-2^k representations. The value $w[j]$ denotes the partial remainder obtained at step j. The quotient after j steps is $q[j] = \sum_{i=1}^{j} q_i r^{-i}$ and the final quotient is $q = q[n]$.

B. Restoring Algorithm

The restoring algorithm uses radix-2 quotient digits in the set $\{0, 1\}$. At each iteration, the algorithm subtracts the divisor d from the partial remainder w (line 4 in Figure 1). If the result is strictly less than 0, the previous value should be restored (line 9 in Figure 1). Usually, this restoration is not performed using an addition, but by selecting the value $2w[j-1]$ instead of $w[j]$, which requires the use of an additional register.

```
1  w[0] ⟵ x
2  for j from 1 to n do
3      w[j] ⟵ 2 × w[j − 1]
4      w[j] ⟵ w[j] − d
5      if w[j] ≥ 0 then
6          q_j ⟵ 1
7      else
8          q_j ⟵ 0
9          w[j] ⟵ w[j] + d
```

Fig. 1. Restoring division algorithm

C. Non-Restoring Algorithm

In order to avoid the cost of restoration in some cases, the previous algorithm can be modified for radix-2 quotient digits in the set $\{-1, 1\}$ instead of $\{0, 1\}$. The new version, called non-restoring division, presented in Figure 2, allows the same small amount of computations at each iteration. The conversion of the quotient from the digit set $\{-1, 1\}$ to the standard set $\{0, 1\}$ can be done *on the fly* by using a simple algorithm (see [1]).

```
1  w[0] ⟵ x
2  for j from 1 to n do
3      w[j] ⟵ 2 × w[j − 1]
4      if w[j] ≥ 0 then
5          w[j] ⟵ w[j] − d
6          q_j ⟵ 1
7      else
8          w[j] ⟵ w[j] + d
9          q_j ⟵ −1
```

Fig. 2. Non-restoring division algorithm

III. SRT Division Algorithm

A. Basic Algorithm

The SRT algorithm, like other digit-recurrence algorithms, is similar to the "paper-and-pencil" method. The main idea in hardware SRT dividers is to use limited comparisons (based on a few most significant bits of d and $w[j]$) to speed-up quotient selection. Hardware SRT dividers are typically of low complexity, utilize small area, but have relatively large latencies. A complete book is devoted to digit-recurrence algorithms [9] but mainly for hardware implementations.

The SRT iteration is based on the computation:

$$w[j+1] \longleftarrow r \times w[j] - q_{j+1} \times d$$

where $w[0] = x$.

At each iteration, the main problem is to determine the new digit of the quotient q_{j+1}. In hardware this is done using a table addressed by a few most significant bits of the divisor d and the partial remainder $w[j]$. The partial remainder is represented using a redundant notation to speedup the subtraction of $q_{j+1} \times d$ from $r \times w[j]$.

In the case of a software implementation, which is our case here, tables for quotient digit selection can not be used (in order to avoid cache misses). Furthermore, a redundant number system is not useful for the partial remainder.

B. Standard High-Radix Algorithm

In our target processors, the rectangle multipliers allow to compute products of one full-width register by one half-width register (i.e., $32 \times 16 \rightarrow 32$). In a high-radix iteration, this kind of multiplier can be efficiently used to perform the computation $q_{j+1} \times d$. The quotient can be written in a high-radix representation: using radix between 2^8 and 2^{16} for instance.

But we need to simplify the new quotient digit selection. The idea of high-radix iterations is to use the first most significant bits of the partial remainder as the new quotient digit. To ensure that this trick can be used, the divisor has to be very close to 1. This is possible after an initial prescaling step. Prescaling consists here in multiplying both x and d by a value M such that the product $M \times d$ is very close to 1. The prescaling value is chosen as an approximation of the reciprocal of d (i.e., $M \approx 1/d$).

IV. High-Radix SRT for VLIW Integer Processor

A. Initial reciprocal approximation

We need M, an initial approximation of the reciprocal such that the value $M \times d$ is very close to 1. In general purpose processors, one uses a look-up table (such as the Itanium processor [4]).

In our case, we will use a polynomial approximation instead. On the interval $[1, 2]$ (which contains the mantissa of d), the following degree-1 polynomial can be used to approximate the function $1/d$ up to 4.54 bits of accuracy:

$$p(d) = 1.457106781 - \frac{1}{2} \times d$$

One has to notice that the 4.54 bits of accuracy are only the theoretical approximation error. The evaluation error sums up. But for this very specific polynomial, the product $1/2 \times d$ is performed exactly w.r.t. the internal format. Then, only the subtraction provides an evaluation error (bounded by half an ulp), that is bounded by half a bit. So the total error on the approximation and evaluation on $p(d)$ leads to 4 bits of accuracy.

For this step we compute the following prescaling:

$$\begin{aligned} M &\longleftarrow p(d) \\ d &\longleftarrow d \times M \\ x &\longleftarrow x \times M \end{aligned}$$

Multiplying both the numerator and the denominator in the division x/d keep the actual value of the quotient. The only potential problem is to ensure that prescaling can be done without overflow.

B. Making d closer to 1

The previous initial approximation only allows to compute up to 4 bits at each step of the recurrence. In order to use a higher radix iteration, we want to get a more accurate approximation of the reciprocal, i.e., to make d closer to 1. One can use a "better" polynomial approximation. Generally, a more accurate approximation requires a high degree polynomial. Hence, it leads to larger computation time.

In our case, we chose to use a part of Goldschmidt's algorithm. From the first approximation, the prescaling step, we have $d = 1 + \epsilon$. One can easily compute the value $1 - \epsilon$. Then, the new approximation of the reciprocal is $d \times (1 - \epsilon) = 1 - \epsilon^2$. From a 4-bit initial approximation, we now have about 8 bits of accuracy for each step.

C. High-radix iterations

From this starting point, we can now proceed with high-radix SRT iterations. Each iteration gives 6 bits of the quotient.

$$w[j+1] \longleftarrow r \times w[j] - q_{j+1} \times d$$

The partial remainder $w[j]$ is represented using two's complement representation. It differs from hardware implementations in which $w[j]$ is represented using a redundant number system in order to speedup the subtraction. d is a natural integer. The quotient digit is a small integer on 8 bits.

V. FLIP: Floating-Point Library for Integer Processor

This work is a part of FLIP a C library developed in the Arénaire team. This library provides the five basic operations: addition, subtraction, multiplication, division and square-root for a quasi-fully compliant single-precision (SP) IEEE 754 FP format [10] (flags are not supported). This library also provides some running modes with relaxed characteristics: no denormal numbers or restricted rounding modes for instance. This library has been developed and validated within a collaboration with STMicroelectronics. The library has been targeted to the VLIW (*very long instruction word*) processor cores of the ST200 family.

Processors of the ST200 family executes up to 4 operations per cycle, with a maximum of one control operation (goto, jump, call, return), one memory operation (load, store, prefetch) and two multiply operations per cycle. All arithmetic instructions operate on integer values, with operands belonging either to the General Register file (64×32-bit), or the Branch Register file (8×1-bit). The multiply instructions are restricted to 16×32-bit on the earlier core variants, but have been recently extended to 32-bit by 32-bit integer and fractional multiplication on the ST231 core. There is no divide instruction but a division step instruction suitable for integer division. In order to reduce the use of conditional branches, the ST200 processor also provides conditional selection instructions.

ACKNOWLEDGMENT

This research was supported by the French *Région Rhône-Alpes* within the *"Arithmétique Flottante pour circuits DSP"* project. The authors would like to thank Christophe Monat from STMicroelectronics for his valuable support with the ST200 environment and Jean-Michel Muller for fruitful discussions.

REFERENCES

[1] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2003.

[2] M. J. Flynn and S. F. Oberman, *Advanced Computer Arithmetic Design*. Wiley-Interscience, 2001.

[3] S. F. Oberman and M. J. Flynn, "Design issues in division and other floating-point operations," *IEEE Transactions on Computers*, vol. 46, no. 2, pp. 154–161, Feb. 1997.

[4] M. Cornea, P. T. P. Tang, and J. Harrison, *Scientific Computing on Itanium-based Systems*. Intel Press, 2002.

[5] S. Oberman and M. Flynn, "Division algorithms and implementations," *IEEE Transactions on Computers*, vol. 46, no. 8, pp. 833–854, Aug. 1997.

[6] D. Menard and O. Sentieys, "Automatic evaluation of the accuracy of fixed-point algorithms," in *Design, Automation and Test in Europe (DATE)*, 2002, pp. 529–537.

[7] P. Faraboschi, G. Brown, J. A. Fisher, G.Desoli, and F. Homewood, "Lx: a technology platform for customizable VLIW embedded processing," in *27th Annual International Symposium on Computer Architecture – ISCA'00*, June 2000.

[8] "HP and STMicroelectronics launch "LX"," `http://www.embedded.com/2000/0010/0010feat6.htm`.

[9] M. D. Ercegovac and T. Lang, *Division and Square-Root Algorithms: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic, 1994.

[10] American National Standards Institute and Institute of Electrical and Electronics Engineers, "IEEE standard for binary floating-point arithmetic," *ANSI/IEEE Standard, Std 754-1985*, 1985.