High-Radix Floating-Point Division Algorithms for Embedded VLIW Integer Processors

Claude-Pierre Jeannerod, Saurabh-Kumar Raina, Arnaud Tisserand

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

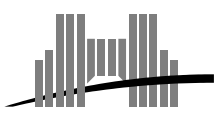
High-Radix Floating-Point Division Algorithms for Embedded VLIW Integer Processors

Claude-Pierre Jeannerod,
Saurabh-Kumar Raina,
Arnaud Tisserand

September 2005

Research Report N° 2005-39

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr



High-Radix Floating-Point Division Algorithms for Embedded VLIW Integer Processors

Claude-Pierre Jeannerod,
Saurabh-Kumar Raina,
Arnaud Tisserand

September 2005

Abstract

This paper 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 FP emulation layer is sometime useful. Here, we focus on high-radix digit-recurrence algorithms for FP division on integer VLIW processors such as the ST200 from STMicroelectronics.

Keywords: Computer arithmetic, floating-point arithmetic, high-radix division algorithm, integer processor, VLIW processor.

Résumé

Ce papier présente quelques algorithmes de division flottante et leur implantation pour des processeurs embarqués entiers de type VLIW. De tels processeurs ne possèdent pas d'unité flottante matérielle mais, pour des raisons de portabilité et/ou de précision, disposer d'une bibliothèque flottante est parfois utile. On se concentre ici sur les algorithmes par addition/décalage, en grande base ; les processeurs visés sont ceux de la famille ST200 de STMicroelectronics.

Mots-clés: Arithmétique des ordinateurs, arithmétique virgule flottante, algorithme de division en grande base, processeur entier, processeur VLIW.

1 Introduction

The implementation of fast floating-point (FP) division is not a trivial task [4, 6]. A study from Oberman and Flynn [8] 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 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 [2]). The main division algorithms and implementations used in general purpose processors can be found in a complete survey [9].

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 [7]. 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, there is no such problem with FP arithmetic.

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.

In this paper, we focus on pure software division implementation based on high-radix SRT (from the initials of Sweeney, Robertson, Tocher) algorithms. We investigate the implementations of these algorithms on processors with rectangular multipliers (e.g., $16\times32 \rightarrow 32$), more or less long latencies for the multiplication unit and parallel functional units (multipliers and ALU).

This work is part of FLIP (*floating-point library for integer processor*) which is a C library for the software support of single precision FP arithmetic on processors without FP hardware units such as VLIW (*Very Long Instruction Word*) or DSP (*Digital Signal Processing*) processor cores for embedded applications. Our current target is the ST200 processor from STMicroelectronics [5].

The paper is organized as follows. Section 2 presents definitions and notations. Section 3 recalls the basic algorithms used for software division. Section 4 presents the standard SRT algorithms (with radix $r = \{2, 4, 16\}$) and their impementations on the ST200 processor. The implementation of the high-radix algorithm on the ST200 processor and its comparison to other algorithms are presented in Section 5. In Section 6, we briefly present the FLIP library and the target ST200 procesor architecture.

2 Definitions and notations

In this paper we follow the definitions and notations of [4]. 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 $d \in [1, 2)$. Hence $q \in [0, 1)$. 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, $k \in \{1, 2\}$. The value $w[j]$ denotes the partial remainder or the residual 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 $q = q[n]$.

3 Basic Software Division

Digit-recurrence algorithms produce a fixed number of quotient bits at each iteration [3, 4] where every iteration produces one quotient digit (one or more quotient bits), most significant quotient digits first. Such algorithms are similar to the "paper-and-pencil" method. It is well known that the choice of radix and quotient digit set influences the overall latency of the algorithm [4]. Roughly, increasing the radix decreases the number of iterations required for the same quotient precision. Unfortunately, as the radix increases, every iteration becomes more complicated and the overall latency is not reduced as expected. Additionally, it becomes impractical to generate the required divisor multiples for higher radices. 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.

3.1 Restoring Algorithm

The restoring algorithm uses radix $r = 2$ with quotient digit set $\{0, 1\}$. At each iteration, the algorithm subtracts the divisor d from the previous partial remainder $w[j-1]$ multiplied by r (line4 in Fig. 1). The quotient digit selection function is derived by comparing the residuals at each step with the divisor multiples (here 0 or d). If the result is strictly less than 0, the previous value should be restored (line 9 in Fig. 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 or conditional execution.

```
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
```

Figure 1: Restoring division algorithm

3.2 Non-Restoring Algorithm

Nonrestoring division [4] is an improved version of the restoring method in the sense that it completely avoids the restoration step by combining restoration additions with the next recurrence, thus, reducing the overall latency. Moreover, it uses the quotient digit set $\{-1, +1\}$ to perform directly the recurrence

with the selection function

$$q_j = \begin{cases} 1 & \text{if } w[j-1] \geq 0, \\ -1 & \text{if } w[j-1] < 0. \end{cases}$$

The nonrestoring division algorithm presented in Fig.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 [4, p. 256]). An extra step enusres a positive final remainder by adding the divisor and consequently modifying the quotient.

```
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
```

Figure 2: Non-restoring division algorithm

The nonrestoring algorithm requires n shifts and n additions/subtractions to obtain a n digit quotient, and is therefore faster than the restoring one (see [3, p. 180]).

4 SRT Division Algorithms for Integer Processors

The SRT algorithm, like other digit-recurrence algorithms, is similar to the "paper-and-pencil" method in the sense that it iteratively computes the digits of the quotient. In radix r and for $j \geq 1$, the jth iteration of the SRT algorithm is

$$w[j] \longleftarrow r \times w[j-1] - q_j \times d,$$

where $w[0] = x$. As before, the main problem is to determine the new quotient digit q_j at each iteration. However, unlike the restoring and non-restoring algorithms of Sec.3, all SRT methods use a *redundant quotient digit set* so as to speed up the computation of q_j. More precisely, comparing the partial remainder to all the divisor multiples can offset or possibly diminish all of the performance gained by increasing the radix. To avoid this, redundancy is introduced in the set of possible quotient digits: in radix r, q_j can be chosen among more than r values [3, p. 10]. This allows to simplify the quotient digit selection function in the sense that comparisons are now done with *limited* precision constants only.

In hardware, the implementation 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 $r \times w[j]$. 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 [3] but mainly for hardware implementations.

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

All the algorithms presented in this paper are tuned to the IEEE 754 standard for binary floating-point arithmetic [1] in *single precision*. Hence x and d are 24-bit significands represented in the normalized floating-point format, that is, they both lie in the range $[1, 2)$.

The implementations we describe below apply only to the significands of the input operands, since the final exponent and sign of the result can be computed easily. Also, since SRT division algorithms use a redundant quotient digit set, it is customary to restrict the range of the divisor d so that $d \in [1/2, 1)$. Consequently, the dividend x should be scaled accordingly so that $x \in (0, d)$. Both scalings are easily done by shifting. In the rest of the paper, we thus assume that $x \in (0, d)$ and $d \in [1/2, 1)$. The goal of FP division is to compute a normalized quotient of 24 bits, and one extra bit (guard bit) for rounding. But sometimes, depending on the algorithm, the precision of the quotient must be extended to one or more bits, which requires one or more extra iterations (see Sec.4.2).

In Sec. 4.1, we recall the quotient digit selection function that is classically used for the radix-2 SRT. In Sec. 4.2, we focus on radix-4 SRT and show how in this case the classical selection function can be optimized for our target architecture. In Sec. 4.3, we recall the basic high-radix SRT division algorithm and the prescaling method.

4.1 Radix-2 SRT division algorithm

In the nonrestoring algorithm, an addition or a subtraction is always performed, whether $q_j = -1$ or $q_j = 1$. Radix-2 SRT division (Fig. 3) uses a redundant quotient digit set $\{-1, 0, 1\}$ and selects $q_j = 0$ that replaces some additions/subtractions by simple shifts. This algorithm uses the iteration

$$w[j] \longleftarrow 2w[j-1] - q_j \times d,$$

and the selection function

$$q_j = \begin{cases} +1 & \text{if } w[j-1] \geq 1/2, \\ 0 & \text{if } -1/2 \leq w[j-1] < 1/2, \\ -1 & \text{if } w[j-1] < -1/2. \end{cases}$$

Notice first that this selection function does *not* depend on d; also, when the partial remainder lies in $[-1/2, 1/2)$ then 0 is selected and thus no operation is performed. Again, as in the non restoring algorithm, the conversion of the quotient from the redundant digit set $\{-1, 0, 1\}$ to the standard digit set $\{0, 1\}$ can be done *on the fly* by using a simple algorithm (see [4, p. 256]). If $w[n] < 0$, the divisor is added to it and the last bit of the quotient is modified accordingly in order to ensure a positive final remainder.

4.2 Radix-4 SRT division algorithm

This SRT algorithm uses radix $r = 4$ and thus retires two bits of the quotient at each iteration. The number of iterations is typically divided by two but, at each iteration, selecting the quotient digit is now more complicated. Unlike for radix-2, the selection function for radix 4 does not solely depend on constants but also on the divisor d. Since the selection function depends on a few bits of the divisor too, partial remainders are compared to the selection constants corresponding to each interval of the divisor. Fig. 5 shows the positive half of the P-D diagram[1] for the radix-4 SRT algorithm with quotient digit set $\{-2, \ldots, 2\}$. This diagram helps to choose the selection constants for each interval in two

[1] P = Partial remainder, D = Divisor [4, p. 283]

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

Figure 3: Radix-2 SRT division algorithm

overlap regions. In the radix-4 case, two possibilities exist for the redundant digit set. This set is either $\{-2,\ldots,2\}$ (in which case it is called *minimally redundant*), or $\{-3,\ldots,3\}$ (in which case it is called *maximally redundant*). The latter has greater redundancy and thus reduces the complexity and latency of the selection function, while the former reduces the number of divisor multiples used for comparisons. The fact that the less redundant set provides smaller overlap regions (see Fig.5) forces to examine the partial remainder $w[j]$ and the divisor d with greater accuracy and hence increases the number of comparisons. Moreover, the partial remainder $w[j]$ must be bounded by $|w[j]| \leq 2/3d$ in case of a minimally redundant set [3, p. 11]. So it is necessary to scale the first partial remainder $w[0] = x$ to the proper range. Consequently, this scaling requires the quotient to be computed with two extra bits of precision, which in turn needs an extra iteration [3, p. 41].

In general, the range of the divisor is determined first, and then a particular set of selection constants is chosen depending on this interval. This determination requires up to three comparisons in the case of the digit set $\{-2,\ldots,2\}$ and at most one in the case of $\{-3,\ldots,3\}$. It turns out that, for our target architecture, these comparisons increase the latency significantly and should be avoided as much as possible. Therefore we suggest below a comparison-free alternative for computing the two selection constants, say s_1 and s_2, corresponding to the digit set $\{-2,\ldots,2\}$. (Note that, when using $\{-3,\ldots,3\}$, there are now three selection constants but they can be obtained with exactly the same approach.) Writing $\lfloor * \rfloor$ for the usual floor function of a real number $*$ and recalling that d has the form $d = 0.1d_2d_3d_4\cdots d_n$ with $d_i \in \{0,1\}$, we define s_1 and s_2 as the following functions of d:

$$s_1 = \frac{3}{4} + \frac{1}{2}d_2 + \frac{1}{4}d_3 + \frac{1}{8}d_4 - t,$$

where

$$t = \frac{1}{8}\left\lfloor d + \frac{3}{16}\right\rfloor,$$

and

$$s_2 = \frac{1}{4} + \frac{1}{8}d_2.$$

For a given d, the values s_1 and s_2 can thus be obtained without any comparison with d. Additionally, by definition of t, one has

$$t = \begin{cases} 0 & \text{if } d \in \left[\frac{1}{2}, \frac{13}{16}\right), \\ 1 & \text{if } d \in \left[\frac{13}{16}, 1\right), \end{cases}$$

```
1  w[0] ⟵ x
2  t ⟵ ⌊d + 3/16⌋ × 2^−3
     where d = 0.1d2d3d4 ··· dn
3  s1 ⟵ 3/4 + 2^−1 d2 + 2^−2 d3 + 2^−3 d4 − t
4  s2 ⟵ 1/4 + d2/8
5  n ⟵ ⌊n/2⌋ + 1
6  for j from 1 to n do
7      w[j] ⟵ 4 × w[j − 1]
8      if w[j] ≥ 0 then
9          if w[j] ≥ s1 then
10             w[j] ⟵ w[j] − 2 × d
11             qj ⟵ 2
12         else
13             if w[j] ≥ s2 then
14                 w[j] ⟵ w[j] − d
15                 qj ⟵ 1
16             else
17                 qj ⟵ 0
18     else
19         if w[j] ≤ −s1 then
20             w[j] ⟵ w[j] + 2 × d
21             qj ⟵ −2
22         else
23             if w[j] ≤ −s2 then
24                 w[j] ⟵ w[j] + d
25                 qj ⟵ −1
26             else
27                 qj ⟵ 0
```

Figure 4: Radix-4 SRT division algorithm with digit set $\{-2, \ldots, 2\}$ for single precision ($n = 24$)

and one may check that s_1 is exactly the "staircase" that belongs to the upper overlap region of the P-D diagram in Fig. 5. Similarly, the "staircase" in the lower overlap region is defined by s_2.

We remark that the value of t actually depends only on $0.1d_2d_3d_4$. However, in our software context, getting these particular bits would require one more logical operation (mask). Computing t with the whole d is thus more efficient.

A radix-4 SRT implementation using the selection constants s_1 and s_2 is given in Fig. 4.

The same kind of selection function has been used for the digit set $\{-3, \ldots, 3\}$. But, as can seen in Table 1, using $\{-2, \ldots, 2\}$ results in a faster implementation. This is due to the fact that there are fewer divisor multiples.

4.3 Standard High-Radix Algorithm

The idea of the high-radix algorithm is to use a higher radix and to select a fixed number of most significant bits of the shifted partial remainder as the quotient digit. In the previous algorithms in Sec. 4.1 and 4.2, the approach of selecting the quotient digit by selection constants is complicated

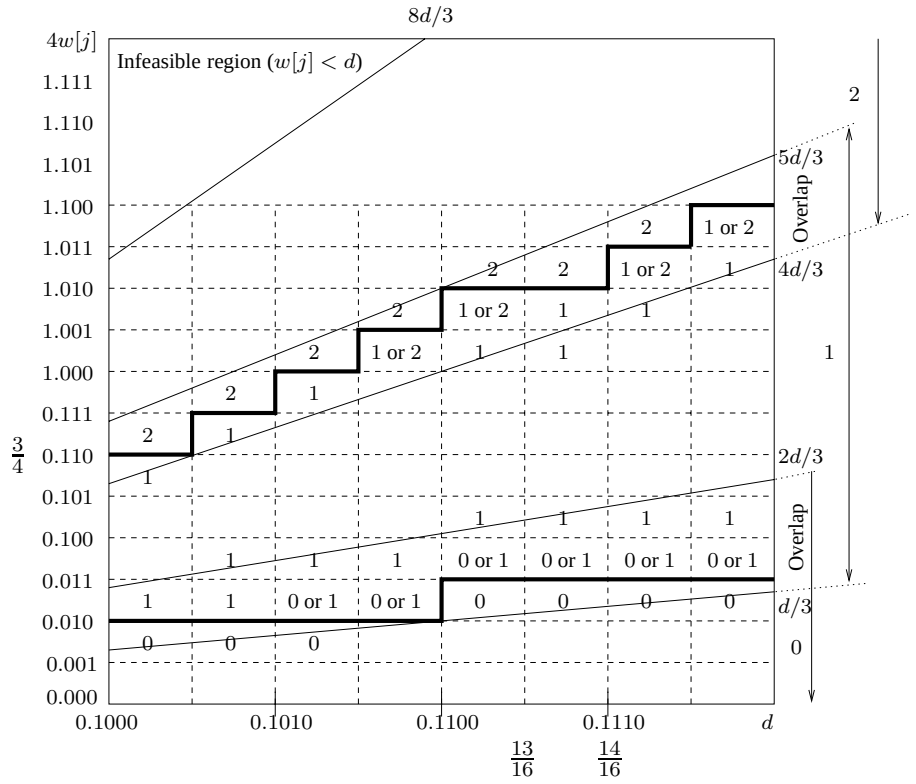


Figure 5: Radix-4 SRT division with quotient digit set $\{-2, \ldots, 2\}$: P-D diagram

due to the large range of the divisor. So, the complexity of the selection function can be reduced by restricting the range of the divisor. Since the overlap is larger when close to $d = 1$, it is convenient to restrict the divisor to a range close to 1. This range restriction can be done by *prescaling the divisor*. Moreover, to preserve the value of the quotient, the dividend has to be prescaled too. Since the range of the divisor is now small, a selection function that is independent of the divisor can be implemented. 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. In general purpose processors such as the Itanium processor [2] a right value for M is looked up in a table. But here we shall define M as $p(d)$ where $p(d)$ is a polynomial that approximates $1/d$ over $[1, 2)$. Now, the product $M \times x$ is very close to the quotient q and therefore a fixed number, say m, of most significant bits of Mx can be used as the first quotient digit. Once M is known, we take $w[0] = M \times x$, $r = 2^m$ and we perform as before iterations like

$$w[j] \longleftarrow r \times w[j-1] - q_j \times d.$$

Since $r = 2^m$ is potentially much larger than 2 or even 4 (typically $8 \leq m \leq 16$, due to the availability of $16 \times 32 \rightarrow 32$ multipliers), each iteration is called a *high-radix iteration*.

For example, let $M = p(d)$ where $p(d)$ is the polynomial of degree 1 given by

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

This gives $1 - 2^{-m} < M \times d < 1 + 2^{-m}$ with $m = 4$ and thus we take $r = 2^4 = 16$. Hence, starting with $w[0] = M$, the full quotient q in single precision is obtained after at most 7 high-radix iterations.

Remark that, unlike the previous SRT algorithms, the product $q_j \times d$ is not computed in a shift-and-add fashion anymore but via a full multiplication. For such multiplications, since $m \leq 16$ it is enough to use the rectangular multipliers $16 \times 32 \to 32$ available on our target architetcure. Finally, quotient-digit selection can be done in a standard way by *rounding the shifted partial remainder* [8, p. 109].

5 High-Radix SRT for VLIW Integer Processors

Here we study SRT in radix $r = 2^9 = 512$ and show how to implement it efficiently on our target VLIW architecture. As usual for high-radix SRT, we proceed in two stages: first, prescaling via polynomial approximation and then performing a small number (in fact, only 2) of high-radix iterations. For each stage, we further explain optimizations that are specific to the architectures under consideration.

5.1 Prescaling via degree 3 polynomials

In order to compute an approximation M of $1/d$ up to 9 bits of accuracy, we use the following two polynomials $u(d)$ and $v(d)$ of degree 3:

$$\begin{aligned} u(d) &= (-0.4354185d + 2.1661051)\, d^2 - 4.0136915d + 3.2828333, \\ v(d) &= (-0.1099684d + 0.7678062)\, d^2 - 2.0034469d + 2.3153857. \end{aligned} \tag{1}$$

We take $M = u(d)$ for $1 \leq d < 3/2$ and $M = v(d)$ for $3/2 \leq d < 2$. Since the two ranges used here are now twice as small as $[1, 2)$, this allows to have relatively low degree polynomials and yet the desired accuracy of 9 bits.

We can exploit the parallelism of the target processor as follows. The comparison of d to $3/2$ and the evaluation of both polynomials are started simultaneously. Once we know the result of the comparison, we finish the evaluation of only the corresponding polynomial. Therefore, the comparison to $3/2$ does not increase the latency.

Note further that these polynomials of degree 3 are not evaluated with the classical Horner scheme, but by using the value of d^2 instead, as can be seen in (1). Again, two linear terms can be evaluated in parallel.

5.2 Radix-512 SRT iterations

Using the prescaling of Sec. 5.1 we set the first partial remainder $w[0]$ to the value $x \times M$ and perform only two iterations in radix $2^9 = 512$:

$$w[j] \longleftarrow 512 \times w[j-1] - q_j \times d, \quad j = 1, 2.$$

The corresponding algorithm is described in Fig. 6.

The quotient q is now given by $q = 0.q_1q_2q_3$ where each q_i is a 9 bit digit. Also, the multiplication $q_j \times d$ is done with an *intrinsic function* (called __st200mulhhs) which performs signed multiplication $16 \times 32 \to 32$ where the result consists of the 32 most significant bits.

For rounding, we use the ROUND function defined as ROUND$(y) = \lfloor y + 2^{-1} \rfloor$.

Finally, the remainder is computed as $x - q \times d$ where x and d are the initial arguments and *not* their prescaled counterparts. Since q has 24 bits and one guard bit, we have to perform 25×24 multiplication. On our target architecture, this rather long multiplication is done by performing four

$16 \times 32 \rightarrow 32$ multiplications on smaller arguments after splitting. As for the subtracting, $q \times d$ from x, this is performed on the 24 most significant bits only.

The speed achieved with this implementation of high-radix SRT is shown in Table 1. It turns out to be faster than all other implementations. In particular, it is more than twice faster than the implementation of the basic restoring algorithm and almost three times faster than the division available from the native STMicroelectronics library.

Table 1 reports the timing in number of bundles (one bundle corresponds to four operations started at one cycle), to compare the different division methods. This is the value of the average case timing (i.e., no special values and no over/under-flow). In our evaluation we have not considered the influence of denormal numbers on the library. Also, all the results have been computed with the rounding mode set to round-to-nearest even. Table 1 also reports the code size (in 32-bit words).

Table 1: Computation time and code size of the different division algorithms.

Division algorithms	time (# bundle)	code size
Original (STMicroelectronics)	171	784
Restoring	134	676
Nonrestoring	117	792
radix-2 SRT	131	820
radix-4 SRT with $\{-2, \ldots, 2\}$	125	716
radix-4 SRT with $\{-3, \ldots, 3\}$	155	628
High-radix SRT	60	840

6 FLIP Library and ST200 processor

This work is a part of FLIP (*floating-point library for integer processor*), a C library developed in the Arénaire team [10]. This library provides the five basic operations: addition, subtraction, multiplication, division and square-root for the single-precision IEEE 754 FP format [1]. This library

```
1  if d < 3/2 then
2      M ⟵ u(d)
3  else
4      M ⟵ v(d)
5  d ⟵ d × M
6  w[0] ⟵ x × M
7  q1 ← ROUND(512 × w[0])
8  w[1] ← 512 × w[0] − q1 × d
9  q2 ← ROUND(512 × w[1])
10 w[2] ← 512 × w[1] − q2 × d
11 q3 ← ROUND(512 × w[2])
```

Figure 6: High-radix SRT division algorithm

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 from STMicroelectronics.

The processor of this family is a four issue VLIW processor and is significantly simpler and smaller than an equivalent four issue super-scalar processor. The compiler is key to generate, schedule and bundle operations, removing the need to add complex hardware to achieve the same results. Fig.7 displays the basic organization of a ST200 processor. Processors of the ST200 family execute up to 4 ALU 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 target core. In order to reduce the use of conditional branches, the ST200 processor also provides conditional selection instructions. Almost all operation latencies are 1 cycle, meaning that the result of an operation can be used in the next *bundle*. Some instructions have a 3-cycle latency (load, multiply, compare to branch) or in one case a 4-cycle latency (load link register LR to call).

7 Conclusion

In this paper, we have presented an implementation of a fast division algorithm for single precison IEEE floating-point arithmetic for processors without FP hardware units such as VLIW or DSP processor cores for the embedded applications. This algorithm has been optimized and tested on STMicroelectronics processors of the ST200 family.

The performances of the new division have been compared to those of the original library. The conclusions of the simulatons done with this architecture are.

- The new division algorithm is close to 3 times faster than the original one.
- The code size of the individual operations is up to 7% larger. But for complete applications, this increase is limited to less than $1\ \%$. So, the instructions cache performances should not be changed.

We aim at extending this algorithm to other highly common algebraic operations such as $1/x$, $1/\sqrt{x}$, or $1/\sqrt{x^2+y^2}$.

Acknowledgments

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] 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.

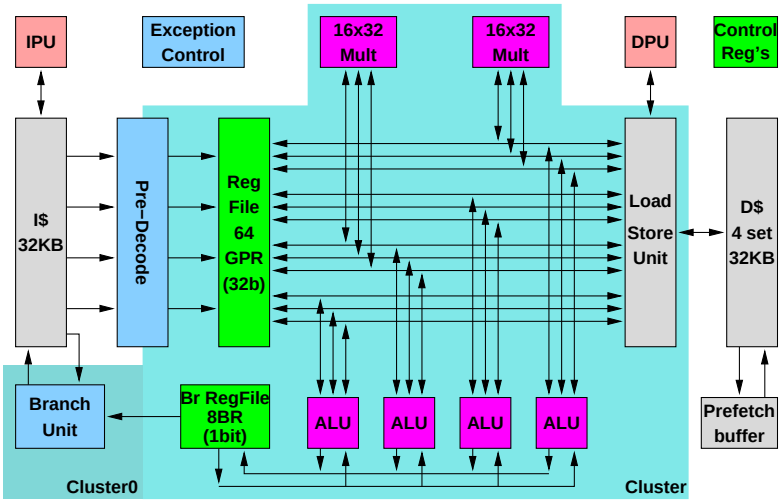


Figure 7: The ST200 VLIW processor architecture.

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

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

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

[5] 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.

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

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

[8] S. F. Oberman and M. J. Flynn. Design issues in division and other floating-point operations. *IEEE Transactions on Computers*, 46(2):154–161, February 1997.

[9] S. F. Oberman and M. J. Flynn. Division algorithms and implementations. *IEEE Transactions on Computers*, 46(8):833–854, August 1997.

[10] Arénaire project/team (computer arithmetic). `http://www.ens-lyon.fr/Arenaire/.`