A new binary floating-point division algorithm and its software implementation on the ST231 processor

Claude-Pierre Jeannerod, Hervé Knochel, Christophe Monat, Guillaume Revy, Gilles Villard

A new binary floating-point division algorithm and its software implementation on the ST231 processor

Claude-Pierre Jeannerod[1, 2] Hervé Knochel[4]
Christophe Monat[4] Guillaume Revy[2, 1] Gilles Villard[3, 2, 1]

[1]INRIA, [2]Université de Lyon, [3]CNRS, [4]STMicroelectronics
Laboratoire LIP, École normale supérieure de Lyon — 46, allée d'Italie, 69364 Lyon cedex 07, France
Compilation Expertise Center, STMicroelectronics — 12, rue Jules Horowitz BP217, 38019 Grenoble cedex, France

Abstract

This paper deals with the design and implementation of low latency software for binary floating-point division with correct rounding to nearest. The approach we present here targets a VLIW integer processor of the ST200 family, and is based on fast and accurate programs for evaluating some particular bivariate polynomials. We start by giving approximation and evaluation error conditions that are sufficient to ensure correct rounding. Then we describe the heuristics used to generate such evaluation programs, as well as those used to automatically validate their accuracy. Finally, we propose, for the binary32 format, a complete C implementation of the resulting division algorithm. With the ST200 compiler and compared to previous implementations, the speed-up observed with our approach is by a factor of almost 1.8.

Keywords: *binary floating-point division, correct rounding, polynomial evaluation, code generation and validation, VLIW integer processor.*

1. Introduction

Although floating-point divisions are less frequent in applications than other basic arithmetic operations, reducing their latency is often an issue [16]. Since low latency implementations may typically be obtained by expressing and exploiting instruction parallelism, intrinsically parallel algorithms tend to be favored. Some examples are [18, 4] for hardware, and [12, 19] for software.

In this paper we focus on designing and implementing low latency floating-point division software for STMicroelectronics' ST231 processor. For this 32-bit VLIW integer processor, various correctly-rounded binary floating-point division algorithms have already been implemented [19]. There, the lowest latency measured is of 48 cycles; it was obtained by exposing instruction-level parallelism (ILP) by means of a suitable combination of univariate polynomial evaluation and Goldschmidt's method, in a way similar to [18]. However, we will see in this paper that much more ILP can in fact be exposed by relying exclusively on polynomial evaluation, leading to a latency of 27 cycles and thus a speed-up by a factor of almost 1.8. To get this result we shall take three main steps, which we summarize now.

As a first step, we extend to division the polynomial-based algorithm introduced in [8] for square rooting. This extension provides a set of approximation and evaluation error conditions that will be proven to be sufficient to ensure correct rounding.

The second step consists in generating an efficient polynomial evaluation program, and in the automatic validation of its accuracy. On the one hand, the generation relies on several heuristics aimed at maximizing ILP exposure. On the other hand, checking that such a program is accurate enough turns out to be more involved than for square root, and will require a specific splitting strategy.

Third, besides the polynomial evaluation program, additional subroutines and associated C code sequences are proposed. Those include sign and exponent computation, rounding, and handling overflow, underflow and special values. This provides a complete division code (for rounding to nearest and without subnormal numbers), optimized for the ST231 processor yet portable.

The paper is organized as follows. After some preliminaries and notation in Section 2, the three contributions above will be described in Sections 3, 4, 5, respectively. Some experimental results obtained with the ST231 processor and the ST200 compiler will be reported in Section 6, and we shall conclude in Section 7.

2. Preliminaries and notation

Floating-point data and rounding. The floating-point data we shall consider are ± 0, $\pm\infty$, quiet or signaling Not-a-Numbers (qNaN, sNaN), as well as *normal* binary floating-point numbers

$$x = (-1)^{s_x} \cdot m_x \cdot 2^{e_x}, \tag{1}$$

with $s_x \in \{0, 1\}$, $m_x = (1.m_{x,1} \dots m_{x,p-1})_2$ and $e_x \in \{e_{\min}, \dots, e_{\max}\}$. (In particular subnormal numbers [1, §3.3] will not be considered.) The precision p and extremal exponents $e_{\min}$ and $e_{\max}$ are assumed to be integers such that $p \geq 2$ and $e_{\min} = 1 - e_{\max}$.

The rounding attribute chosen here is "to nearest even" (roundTiesToEven [1, §4.3.1]) and will be referred to as RN. Except for some special input (x, y) for which special values must be delivered as detailed in Section 5.4, the standard requires that $\mathrm{RN}(x/y)$ be returned whenever x and y are as in (1).

Correctly rounded division. That $\mathrm{RN}(x/y)$ essentially reduces to a correctly rounded ratio of (possibly scaled) significands can be recalled as follows. First, using $\mathrm{RN}(-x) = -\mathrm{RN}(x)$ gives

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

where s_r is the XOR of s_x and s_y. Then, taking $c = 1$ if $m_x \geq m_y$, and 0 otherwise, one has $|x/y| = \ell \cdot 2^d$, where

$$\ell = 2m_x/m_y \cdot 2^{-c}, \quad d = e_x - e_y - 1 + c. \tag{2}$$

Since both m_x and m_y are in $[1, 2)$, ℓ is in $[1, 2)$ as well, and $\ell \cdot 2^d$ will be called the *normalized representation* of $|x/y|$. Tighter enclosures of ℓ can in fact be given:

Property 1. *If $m_x \geq m_y$ then $\ell \in [1, 2 - 2^{1-p}]$ else $\ell \in (1, 2 - 2^{1-p})$.*

Proof. If $m_x \geq m_y$ then $c = 1$, and we deduce from $1 \leq m_x \leq 2 - 2^{1-p}$ and $0 < 1/m_x \leq 1/m_y \leq 1$ that $1 \leq \ell \leq 2 - 2^{1-p}$. If $m_x < m_y$ then $m_x \leq m_y - 2^{1-p}$ and thus $\ell \leq 2 - 2^{2-p}/m_y$. Hence, using $m_x \geq 1$ and $1/m_y > 1/2$, we obtain $1 < \ell < 2 - 2^{1-p}$ as desired. □

These enclosures of ℓ will be used explicitly when handling underflow in Section 5.3. For now, we simply note that both of them give $\mathrm{RN}(\ell) \in [1, 2 - 2^{1-p}]$, so that

$$\mathrm{RN}(|x/y|) = \mathrm{RN}(\ell) \cdot 2^d.$$

If $e_{\min} \leq d \leq e_{\max}$ then we shall return the normal number $\mathrm{RN}(x/y) = (-1)^{s_r} \cdot m_r \cdot 2^{e_r}$, where

$$s_r = s_x \oplus s_y, \quad m_r = \mathrm{RN}(\ell), \quad e_r = d. \tag{3}$$

Else, d is either smaller than $e_{\min}$ or greater than $e_{\max}$. Since subnormals are not supported, some special values will be returned in either case, as detailed in Section 5.3. Thus, to get $\mathrm{RN}(x/y)$ the main task consists in deducing from m_x and m_y the correctly rounded value $\mathrm{RN}(\ell)$ in (3), the sign s_r and exponent e_r being computable in parallel and at lower cost (see Section 5.1).

Correct rounding from one-sided approximations. Among the many methods known for getting $\mathrm{RN}(\ell)$ (see [5, §8.6] and the references therein), the one we focus on in this paper uses *one-sided approximations*: as shown in [5, p. 459], this method reduces the computation of $\mathrm{RN}(\ell)$ to that of an approximation v of ℓ such that

$$-2^{-p} < \ell - v \leq 0. \tag{4}$$

Here v is representable with, say, k bits while ℓ has in most cases an infinite binary expansion $(1.\ell_1 \dots \ell_{p-1}\ell_p \dots)_2$.

Once such a v is known, correct rounding follows easily (see [5, p.460] and Section 5.2) and it remains to deduce from each possible pair (m_x, m_y) a value v that satisfies (4).

Before presenting in Sections 3 and 4 a novel approach for computing v in a certified and efficient way, we give next a brief description of the implementation context.

Software implementation on integer processors. Our implementation of division is for the binary32 format of [1]:

$$k = 32, \quad p = 24, \quad e_{\max} = 127.$$

It will consist of a piece of C code using exclusively 32-bit integers, for input/output encoding as well as for intermediate variables.

Concerning the input data x, y the standard encoding into 32-bit unsigned integers X, Y is assumed [1, §3.4]. For example, the bit string $X_{31} \dots X_0$ of $X = \sum_{i=0}^{31} X_i 2^i$ is such that $X_{31} = s_x$ (sign bit of x), $\sum_{i=0}^{7} X_{i+23} 2^i = e_x + 127$ (biased exponent of x, for x normal), and $X_i = m_{x,23-i}$ for $i = 0, \dots, 22$ (fraction bits of x). The output is encoded similarly and our division code eventually takes the form of a C function like the one below:

```
unsigned int binary32div( unsigned int X ,
                          unsigned int Y ) {...}
```

Concerning the operations on input or intermediate variables, we assume available the basic arithmetic and logical operators `+`, `&`, etc. as well as a `max` instruction and a multiplier `mul` defined for $A, B \in \{0, \dots, 2^{32} - 1\}$ as $\mathtt{mul}(A, B) = \lfloor A \cdot B \cdot 2^{-32} \rfloor$. Here $\lfloor \cdot \rfloor$ denotes the usual floor function, and `mul` thus gives the 32 most significant bits of the exact product $A \cdot B$.

Therefore, up to emulating `max` and `mul` as shown in Appendix A, all we need is a C compiler that implements 32-bit arithmetic. However some features of the ST231 processor and compiler have influenced the design and optimizations described in Sections 4 and 5. In particular, 4

instructions can be launched simultaneously, among which at most two `mul` instructions. Also, the latency of `max` and of the other basic operators is 1, while it is 3 for `mul`.

3. Sufficient conditions for correct rounding

This section gives sufficient conditions for (4) to hold. To do so we extend to division the bivariate polynomial-based framework introduced in [8] for computing correctly-rounded floating-point square roots. A preliminary step is to restrict (4) to the following more symmetric, but only slightly stronger, condition

$$\left|(\ell + 2^{-p-1}) - v\right| < 2^{-p-1}. \tag{5}$$

Sufficient conditions to have (5) will be obtained by introducing a suitable bivariate polynomial approximant along with approximation and evaluation error bounds. We will rely on these conditions for designing our validation approach in Section 4.2, especially for providing hint values to the software tools *Sollya* and *Gappa* that we use.

3.1 Polynomial approximation

We start by interpreting the rational number $\ell + 2^{-p-1}$ as the value of a suitable function of m_x and m_y. Defining $F(s,t) = 2^{-p-1} + s/(1+t)$, we have that $\ell + 2^{-p-1}$ is the exact value of F at the particular point

$$(s^*, t^*) = (2m_x \cdot 2^{-c}, m_y - 1). \tag{6}$$

When m_x and m_y vary then s^* and t^* range in the domains

$$\mathcal{S} = [1, 2-2^{1-p}] \cup [2, 4-2^{3-p}] \quad \text{and} \quad \mathcal{T} = [0, 1-2^{1-p}].$$

(In particular, the second interval for $\mathcal{S}$ comes from the fact that $c = 0$ implies $m_x \leq 2 - 2^{2-p}$.)

Then, following [8], the next step is to approximate F over $\mathcal{S} \times \mathcal{T}$ by a suitable bivariate polynomial P. Since F is linear with respect to s, one can reduce to univariate approximation by taking

$$P(s,t) = 2^{-p-1} + s \cdot a(t), \tag{7}$$

with $a(t)$ a polynomial approximating $1/(1+t)$ over $\mathcal{T}$. If we define

$$\alpha(a) = \max_{t \in \mathcal{T}} \left|1/(1+t) - a(t)\right|, \tag{8}$$

then the *approximation error* for F at (s^*, t^*) satisfies

$$|F(s^*, t^*) - P(s^*, t^*)| \leq \left(4 - 2^{3-p}\right) \alpha(a).$$

Since from (5) the overall error must be less than 2^{-p-1}, we take

$$\alpha(a) < 2^{-p-1} / \left(4 - 2^{3-p}\right) \tag{9}$$

as a target approximation error bound for computing the approximant a.

The approximant construction is done using the software environment *Sollya*[1], from the function $1/(1+t)$, the domain $\mathcal{T}$, and the error bound (9). For reducing the final cost we choose a polynomial of smallest degree δ that satisfies the above constraints. For $p = 24$, the *Sollya* function `guessdegree` leads to $\delta = 10$. Then, with the additional constraint that the coefficients should have no more than $k = 32$ fraction bits, we deduce a suitable polynomial $a(t)$ from the `remez` function and by truncation. One obtains $a(t) = \sum_{i=0}^{10} a_i t^i$, with each $|a_i|$ defined by a 32-bit unsigned integer A_i such that $|a_i| = A_i \cdot 2^{-32}$. In our case, it turns out that $1 > |a_0| > \cdots > |a_{10}|$ and that the signs alternate, so that

$$a_i = (-1)^i \cdot A_i \cdot 2^{-32} \in (-1, 1), \quad 0 \leq i \leq 10. \tag{10}$$

Finally, a certified bound on the actual approximation error is obtained using *Sollya*'s `infnorm` function:

$$\alpha(a) \leq 3 \cdot 2^{-29} \approx 2^{-27.41} < \frac{2^{-25}}{4 - 2^{-21}} \approx 2^{-27}. \tag{11}$$

3.2. Subdomain-based error conditions

Once a is available, for establishing (5) we need to consider the *rounding errors* of the evaluation of P given by (7). In [8], for the design of a correctly-rounded square root function in a similar context, errors have been bounded by a single value, on the whole domain $\mathcal{S} \times \mathcal{T}$. Here we extend this approach since the room left by (11) between the actual approximation error and 2^{-p-1}, is not sufficient for taking into account rounding errors. (We refer to the experimental data in Section 4.2.) The validation of our division algorithm will require to see the interval $\mathcal{T}$ as a union of n subintervals: $\mathcal{T} = \bigcup_{i=1}^{n} \mathcal{T}^{(i)}$ and, accordingly, the approximation error $\alpha(a)$ of (8) will then split up into

$$\alpha^{(i)}(a) = \max_{t \in \mathcal{T}^{(i)}} \left|1/(1+t) - a(t)\right|, \quad 1 \leq i \leq n. \tag{12}$$

The value v in (5) will result from the evaluation of P by a finite precision program $\mathcal{P}$ that produces, for each subdomain $\mathcal{S} \times \mathcal{T}^{(i)}$, the rounding error

$$\rho^{(i)}(\mathcal{P}) = \max_{(s,t) \in \mathcal{S} \times \mathcal{T}^{(i)}} \left|P(s,t) - \mathcal{P}(s,t)\right|. \tag{13}$$

Let i be such that (s^*, t^*) belongs to $\mathcal{S} \times \mathcal{T}^{(i)}$. Then, from (12) and (13), the overall error is eventually bounded as $\left|(\ell + 2^{-p-1}) - v\right| \leq \left(4 - 2^{3-p}\right) \alpha^{(i)}(a) + \rho^{(i)}(\mathcal{P})$, and we arrive at the following sufficient conditions for (5):

[1] `http://sollya.gforge.inria.fr/` and [10].

Property 2. *If the approximation and rounding errors satisfy, for* $1 \leq i \leq n$*,*

$$(4 - 2^{3-p})\,\alpha^{(i)}(a) + \rho^{(i)}(\mathcal{P}) < 2^{-p-1} \qquad (14)$$

then (5) holds.

4. Evaluation code generation and validation

Here we present our strategy to produce automatically an efficient evaluation program $\mathcal{P}$ along with an a posteriori certificate on its accuracy (in the sense of Property 2).

For $p = 24$, the program $\mathcal{P}$ will implement a finite precision evaluation of $P(s^*, t^*) = 2^{-25} + s^* \cdot a(t^*)$. Here a is the polynomial obtained in Section 3.1 and whose coefficients a_i satisfy (10). The program will in fact store only the 32-bit integers A_i, the coefficient signs being handled through an appropriate choice of arithmetic operators ($+$ or $-$) made when generating $\mathcal{P}$. Concerning the evaluation point (s^*, t^*) defined in (6), it can be encoded by a pair (S, T) of 32-bit unsigned integers such that

$$s^* = S \cdot 2^{-30}, \qquad t^* = T \cdot 2^{-32}. \qquad (15)$$

The computation of S and T from X and Y can be implemented in C as follows:

```
Tx = X << 9;        T = Y << 9;        X8 = X << 8;
c  = Tx >= T;  Mx = X8 | 0x80000000;
S = Mx >> c;
```

The above code has been written with ILP exposure in mind. For example, c is computed by comparing the trailing significands $m_x - 1$ and $m_y - 1$, rather than m_x and m_y. On ST231, this piece of code will typically take 3 cycles, the delay between S and T being of 2 cycles.

Concerning the output of $\mathcal{P}$, note first that by Property 1, v in (4) must satisfy $v \in [1, 2-2^{-p})$. Moreover, because of the formats introduced in (10) and (15) and of the fact that $\mathcal{P}$ is essentially a fixed-point code, the output of $\mathcal{P}$ will be a 32-bit unsigned integer V representing v as

$$v = V \cdot 2^{-30}. \qquad (16)$$

(How to implement correct rounding using this format will be detailed in Section 5.2.)

4.1. Evaluation program generation

Once the integers $S, T, A_0, A_1, \ldots, A_{10}$ are available, we determine automatically an efficient program $\mathcal{P}$ for evaluating V by means of 32-bit additions/subtractions and `mul` instructions. By efficient program we mean a way of parenthesizing the arithmetic expression $P(s^*, t^*)$ that reduces the execution latency as well as the number of `mul` instructions. This depends on a set of heuristics, still under development, but that already enabled to produce an evaluation program suitable for floating-point division on ST231.

Evaluation tree generation. Following [6], we generate evaluation trees whose height is kept low through ILP exposure, thus avoiding highly sequential schemes like Horner's. To do so, we proceed in two substeps. Given the polynomial degree δ, the delay between S and T, and some latencies for addition/subtraction and `mul`, we first heuristically compute a target latency τ for $\mathcal{P}$, assuming unbounded parallelism. This hint for the latency is a priori feasible, still reasonably low. Then we generate automatically a set of evaluation trees with height no more than this target latency. An example of such a tree is given in Figure 1, whose height corresponds to the target latency $\tau = 14$. This latency[2] is more than three times lower than the latency of $(3+1) \times 11 = 44$ cycles that would result from computing $P(s^*, t^*)$ with Horner's rule.

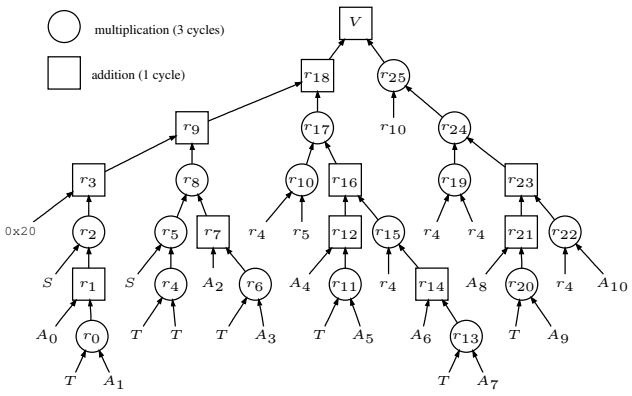

Figure 1. Generated evaluation tree.

Our approach is heuristic in the sense that if no evaluation tree is found that satisfies the target latency τ, we increase τ and restart the process. However, the practice has shown that the number of evaluation trees found given the target τ is usually extremely large, already for degrees much lower than the degree $\delta = 10$ we have here. Consequently, we have implemented several filters in order to reduce significantly this number during the generation, and thus to speed up the whole process.

Arithmetic operator choice. A first filter consists in restricting to evaluation trees for which all intermediate values are positive. This restriction allows to work with the full precision $k = 32$, instead of loosing one bit because of signed arithmetic. Such special trees can be found by choosing the addition/subtraction operators appropriately, for example considering $a_0 - (-a_1 t)$ rather than $a_0 + a_1 t$,

[2] On the tree, it corresponds to computing r_4, and then r_{22} up to V in $3+3+1+3+3+1 = 14$ cycles.

for a_1 is negative. If the sign of one of the intermediate values computed by the tree changes when the input (S, T) varies, then that evaluation tree is rejected. This first filter is implemented using the MPFI[3] library for interval arithmetic. (An interval containing zero is interpreted as a sign change for the corresponding variable.)

Scheduling verification. A second filter consists in checking if a given evaluation tree can be scheduled without latency increase on a simplified model of the real target architecture. In our case of division on ST231, the evaluation tree has a latency of 14 cycles in theory (that is, assuming unbounded parallelism) as well as in practice. In fact, the possible scheduling displayed in Table 1 uses only 3 out of the 4 issues offered by the ST231. For now, this second filter is implemented using a naive list scheduling algorithm.

	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			

Table 1. Feasible scheduling on ST231.

Evaluation code. Finally, the first evaluation tree that passes both filters is chosen, and the corresponding evaluation program is printed out as a piece of C code. In our case, the obtained evaluation program is presented in Listing 1.

```
unsigned int r0  = mul( T , 0xfffffe7d7 );
unsigned int r1  = 0xfffffffe8 - r0;
unsigned int r2  = mul( S , r1 );
unsigned int r3  = 0x00000020 + r2;
unsigned int r4  = mul( T , T );
unsigned int r5  = mul( S , r4 );
unsigned int r6  = mul( T , 0xffbad86f );
unsigned int r7  = 0xfffbece7 - r6;
unsigned int r8  = mul( r5 , r7 );
unsigned int r9  = r3 + r8;
unsigned int r10 = mul( r4 , r5 );
unsigned int r11 = mul( T , 0xf3672b51 );
unsigned int r12 = 0xfd9d3a3e - r11;
unsigned int r13 = mul( T , 0x9a3c4390 );
unsigned int r14 = 0xd4d2ce9b - r13;
unsigned int r15 = mul( r4 , r14 );
unsigned int r16 = r12 + r15;
unsigned int r17 = mul( r10 , r16 );
unsigned int r18 = r9 + r17;
unsigned int r19 = mul( r4 , r4 );
unsigned int r20 = mul( T , 0x1bba92b3 );
unsigned int r21 = 0x525a1a8b - r20;
unsigned int r22 = mul( r4 , 0x0452b1bf );
unsigned int r23 = r21 + r22;
unsigned int r24 = mul( r19 , r23 );
unsigned int r25 = mul( r10 , r24 );
unsigned int V   = r18 + r25;
```

Listing 1. Generated evaluation program.

4.2. Evaluation program validation

We have validated the above evaluation code using *Gappa*[4]. *Gappa* is a software tool intended to help verifying and formally proving properties on numerical programs. It manipulates logical formulas involving the inclusion of expressions in intervals, and allows to bound rounding errors in a certified way. The first validation step is to check that no overflow can occur. This is easily done with *Gappa* by verifying that no intermediate value can be greater than $2^{32} - 1$.

For proving the correct rounding inequality (5), we use Property 2 for providing hints to *Gappa*. With the polynomial a computed in Section 3.1, and no subdivision of the domain $\mathcal{T}$ (that is, $n = 1$), the approximation error satisfies

$$\alpha(a) \leq \theta_0 = 3 \cdot 2^{-29} \approx 2^{-27.41}. \tag{17}$$

According to (14), we then take $\eta_0 = 2^{-25} - (4 - 2^{-21}) \cdot \theta_0$, and use the hint

$$\rho(\mathcal{P}) < \eta_0 \approx 2^{-26.99} \tag{18}$$

as a sufficient condition on the rounding error during the evaluation of the program $\mathcal{P}$. This approach succeeds in the case $m_x \geq m_y$ where, with the help of *Gappa*, we have checked that (18) is true. Note that the strict inequality is checked with *Gappa* through an inequality $\rho(\mathcal{P}) \leq \eta_0 - \epsilon$ for a small enough, positive ϵ.

In the case $m_x < m_y$, with $s^* \in [2, 4 - 2^{-21}]$ and $t^* \in [2^{-23}, 1 - 2^{-23}]$, we need to split up the domain $\mathcal{T}$. Indeed, we find points t^* at which (18) is not satisfied. For example, for $t^* = 0.9749044179916381835937 5$, with t^* in a small enough interval $\mathcal{T}^{(i)}$ (see the last row of Table 2), we can check that $\rho(\mathcal{P}) < \rho^{(i)}(\mathcal{P}) < \eta_4 \approx 2^{-26.77}$. Nevertheless, considering the approximation error on the same interval, we can establish (14) since, in compensation, it can be proven that $\alpha^{(i)}(a) \leq \theta_4 \approx 2^{-27.49} < \alpha(a)$.

For validating the division program, and finding adequate subintervals $\mathcal{T}^{(i)}$'s, we have implemented a splitting of $\mathcal{T}$ by dichotomy. This process has split the definition domain up into $n = 36127$ subintervals. For each of them, using *Sollya*, we first bound the approximation error $\alpha^{(i)}(a)$. This gives $\alpha^{(i)}(a) \leq \theta$ for some rational number θ. Then the sufficient condition (14) is established by checking that $\rho^{(i)}(\mathcal{P}) < \eta = 2^{-25} - (4 - 2^{-21}) \cdot \theta$. The dichotomy is illustrated by Table 2 where "no" in the last column means that the interval in the second column has to be split up.

[3] http://gforge.inria.fr/projects/mpfi/

[4] http://lipforge.ens-lyon.fr/www/gappa/ and [14].

Depth	Subintervals	$\alpha^{(\cdot)}(a) \leq$	$\rho^{(\cdot)}(\mathcal{P}) <$	Does (14) hold?
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
...				
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
	$\mathrm{I}_{j,19309} = [0.921875, 0.9257811307907104492187 5]$	$\theta_3 \approx 2^{-27.44}$	$\eta_3 \approx 2^{-26.90}$	yes
	$\mathrm{I}_{j,19533} = [0.97490406036376953125, 0.97490441799163818359375]$	$\theta_4 \approx 2^{-27.49}$	$\eta_4 \approx 2^{-26.77}$	yes

Table 2. Splitting steps.

The bounds on $\alpha^{(\cdot)}(a)$ and $\rho^{(\cdot)}(\mathcal{P})$ (where the $\cdot$ stands for the index of the interval in the subdivision) that are discovered with *Sollya* and *Gappa* are given in the third and fourth columns. For the exact values of the θ_j's and η_l's we refer to Appendix B.

5. Implementation of a complete division code

So far we have presented an efficient way of deducing from m_x and m_y an integer V such that $v = V \cdot 2^{-30} = (1.v_1 \dots v_{30})_2$ satisfies (4). To obtain a complete code for binary floating-point division it remains to compute the sign and the exponent of the result, to deduce from v a correctly-rounded significand, and to pack those three fields according to the standard encoding of the binary32 format. If either x or y is a special operand then a special value must be returned, as prescribed in [1], which requires specific handling. The following sections detail algorithms for each of those tasks along with some C codes well-suited for a target like the ST231 processor. (All the variables are `unsigned int`, except for D which is `int`.)

From Section 5.1 to Section 5.3, both x and y are supposed to be normal numbers, while in Section 5.4 at least one of them is special, that is, ± 0, $\pm\infty$, qNaN or sNaN.

5.1. Sign and exponent computation

The sign s_r of the result r is trivially obtained by taking the XOR of the sign bits of X and Y:

```
Sr = ( X ^ Y ) & 0x80000000;
```

The result exponent e_r will be obtained by first computing the integer

$$D = d + e_{\max} - 1. \tag{19}$$

If $e_{\min} \leq d \leq e_{\max}$ then r is normal and the bits of its biased exponent $E_r = D + 1$ will be deduced by adding the correctly-rounded significand to $D \cdot 2^{p-1}$ (see Section 5.2); otherwise, since subnormals are not supported, r must take particular values like ± 0, $\pm 2^{e_{\min}}$ or $\pm\infty$, and we will see in Section 5.3 how such situations can be detected directly by inspecting the integer D.

Let us now compute D. Since x and y are normal, their biased exponents are $E_x = e_x + e_{\max}$ and $E_y = e_y + e_{\max}$. Therefore, using (19) together with the definition of d in (2),

$$D = E_x - E_y + e_{\max} - 2 + c. \tag{20}$$

When $k = 32$, $p = 24$, and $e_{\max} = 127$, an implementation of (20) that exposes ILP is:

```
absX = X & 0x7FFFFFFF;      absY = Y & 0x7FFFFFFF;
Ex = absX >> 23;                  Ey = absY >> 23;
int D = (Ex + 125) - (Ey - c);
```

Note that the biased exponent values E_x and E_y have been obtained by first computing the absolute values of x and y (by setting the sign bits to zero), and then shifting right by $p - 1 = 23$ positions. Of course, there are other ways of extracting these biased exponent values, such as, for example for E_x,

```
Ex = ( X >> 23 ) & 0xFF;
```

or

```
Ex = ( X << 1 ) >> 24;
```

However, we tend to prefer the version involving `absX` and `absY`, for these absolute values will be reused when computing special values in Section 5.4.

5.2. Rounding and packing

Given v that approximates ℓ from above as in (4), the correctly rounded significand $m_r = \mathrm{RN}(\ell)$ can be deduced essentially as in [8] and [5, p. 459] (see also [20, 15] and [17, §3.3]): defining

$$w = (1.v_1 \dots v_{p-1}v_p)_2$$

as the truncated value of v to p fraction bits, one has

$$m_r = \begin{cases} w \text{ truncated to } p-1 \text{ fraction bits if } w \geq \ell, \\ w + 2^{-p} \text{ truncated to } p-1 \text{ fraction bits if } w < \ell. \end{cases}$$

To check this, note first that by truncation to p fraction bits, $0 \le v - w < 2^{-p}$, which together with (4) leads to

$$|\ell - w| < 2^{-p}. \tag{21}$$

Then it remains to verify for $v_p = 0$ and for $v_p = 1$, that the above definition of m_r always gives RN(ℓ), using in particular the classic fact that ℓ cannot be exactly halfway between two consecutive normal floating-point numbers (see for example [11, Lemma 1] or [3, p. 229]).

To implement the above definition of m_r one has to compute w and to decide whether $w \ge \ell$. Since $v = V \cdot 2^{2-k}$ and the bit string of V is $01v_1 \dots v_{k-2}$, truncating v to p fraction bits means zeroing out the last $k-p-2$ bits of V. For $(k,p) = (32,24)$ the line below sets to zero the last six bits of the bit string $01v_1 \dots v_{24}v_{25}v_{26}v_{27}v_{28}v_{29}v_{30}$ of V:

```
W = V & 0xFFFFFFC0;
```

The bit string of W is thus $01v_1 \dots v_{24}000000$ and $w = W \cdot 2^{-30}$. To know if $w \ge \ell$ or not, let us introduce the integer M_y such that $m_y = M_y \cdot 2^{1-k}$. With $k = 32$ and S as in (15), we have the following characterization:

Property 3. *The condition $w \ge \ell$ is true if and only if the condition* `mul(W,My) >= (S >> 1)` *is true.*

Proof. From $w = W \cdot 2^{-30}$, $m_y = M_y \cdot 2^{-31}$, and $s^* = S \cdot 2^{-30}$, we deduce that $w \ge \ell = s^*/m_y$ is equivalent to $W \cdot M_y \cdot 2^{-32} \ge S/2$. Now, by definition of `mul` as a floor function, we have

$$W \cdot M_y \cdot 2^{-32} - 1 < \texttt{mul}(W, M_y) \le W \cdot M_y \cdot 2^{-32}. \tag{22}$$

Using the two inequalities in (22) together with the fact that $\texttt{mul}(W, M_y)$ and $S/2$ are nonnegative integers representable with 32 bits, one can check that $w \ge \ell$ if and only if $\texttt{mul}(W, M_y) \ge S/2$. In C, the latter condition reads `mul(W,My) >= (S >> 1)`. □

To deduce the value of $m_r = (1.m_{r,1} \dots m_{r,p-1})_2$ let M_r be the integer such that $m_r = M_r \cdot 2^{1-p}$. It then follows from $w = W \cdot 2^{2-k}$ that

$$M_r = \begin{cases} \lfloor W \cdot 2^{-(k-p-1)} \rfloor & \text{if } w \ge \ell, \\ \lfloor (W + 2^{k-p-2}) \cdot 2^{-(k-p-1)} \rfloor & \text{if } w < \ell. \end{cases}$$

Thus, for $(k,p) = (32,24)$, M_r is obtained by shifting either W or $W + 2^6$ to the right by seven positions.

Let us now pack the bits of m_r with the bits of s_r and E_r. Since in Section 5.1 we have in fact computed $D = E_r - 1$, removing the leading 1 in $m_r = (1.m_{r,1} \dots m_{r,p-1})_2$ can be avoided: the k-bit integer that encodes the result r is

$$s_r \cdot 2^{k-1} + D \cdot 2^{p-1} + M_r. \tag{23}$$

In particular $D \in \{0, \dots, 2e_{\max} - 1\}$ implies that $D \cdot 2^{p-1} + M_r$ must be less than 2^k and thus never propagates a carry to the sign bit.

Since we expect the computation of s_r and D to be much cheaper than that of M_r, one may first take the bitwise OR of $s_r \cdot 2^{k-1}$ and $D \cdot 2^{p-1}$, and then only add M_r. For $(k,p) = (32,24)$ and $p = 24$ this gives the following code sequence:

```
My = (Y << 8) | 0x80000000;
if( mul(W,My) >= (S >> 1) )
   return (Sr | (D << 23)) + (W >> 7);
else
   return (Sr | (D << 23)) + ((W + 0x40) >> 7);
```

5.3. Overflow and underflow detection

The value of the integer $D = E_r - 1$ in (20) can be used to detect overflow and underflow. Indeed, the normality assumption on x and y implies that both E_x and E_y lie in the normal (biased) exponent range $\{1, \dots, 2e_{\max}\}$. Since c is either 0 or 1, it follows from (20) that D ranges from $-e_{\max} - 1$ to $3e_{\max} - 2$. This range contains $\{0, \dots, 2e_{\max} - 1\}$, for which the result is a normal number; it also contains two extremal ranges, for which either overflow or underflow occurs.

Let us start with overflow. If $D \ge 2e_{\max}$ then $E_r = e_r + e_{\max}$ gives $e_r \ge e_{\max} + 1$. Therefore $|x/y| \ge 2^{e_{\max}+1}$ and, by definition of the rounding operator RN($\cdot$) in [1], we have

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

For $e_{\max} = 127$ this case can be implemented as:

```
if( D >= 0xFE ) return Sr | 0x7F800000;
```

Now for underflow. If $D < 0$ then $e_r \le e_{\min} - 1$. Using $\ell < 2$ then gives $|x/y| < 2^{e_{\min}}$. Following [7] we round $|x/y|$ to nearest-even as if subnormals were supported. This rounded value can be either $2^{e_{\min}}$ (the smallest positive normal number) or a subnormal number. By definition of RN($\cdot$) the first case occurs if and only if

$$(1 - 2^{-p}) \cdot 2^{e_{\min}} \le |x/y| < 2^{e_{\min}}, \tag{24}$$

and we shall return the normal number $(-1)^{s_r} 2^{e_{\min}}$. In the second case, since we do not support subnormals, we shall return $(-1)^{s_r} 0$.

Property 4. *One has (24) if and only if $e_x - e_y = -e_{max}$ and $m_x = 2 - 2^{1-p}$ and $m_y = 1$.*

Proof. Assume that (24) holds. Then it follows from $|x/y| = \ell \cdot 2^d$ with $\ell \in [1,2)$ that d must be equal to $e_{\min} - 1 = -e_{\max}$, and that $\ell \ge 2 - 2^{1-p}$. Property 1 implies further that $\ell = 2 - 2^{1-p}$ and $m_x \ge m_y$. Using the definition of ℓ in (2) then gives $m_x/m_y = 2 - 2^{1-p}$ and, since

$1 \le m_x, m_y \le 2 - 2^{1-p}$, we conclude that the only possible value for (m_x, m_y) is $(2 - 2^{1-p}, 1)$. Applying (2) with $(c, d) = (1, -e_{\text{max}})$ gives furthermore $e_x - e_y = -e_{\text{max}}$. Conversely, one may check that if $(m_x, m_y) = (2 - 2^{1-p}, 1)$ and $e_x - e_y = -e_{\text{max}}$ then $|x/y| = (1 - 2^{-p}) \cdot 2^{e_{\text{min}}}$. □

Using (20) with $c = 1$ shows that the condition $e_x - e_y = -e_{\text{max}}$ is equivalent to $D = -1$. With M_x and M_y computed as in Sections 4 and 5.2, Property 4 thus suggests the following code for handling underflow:

```
if( D < 0 ) {
   if( D == -1 && Mx == 0xFFFFFF00
                    && My == 0x80000000 )
      return Sr | 0x00800000;
   else
      return Sr;
}
```

5.4. Computing special values

Assume now that (x, y) is a *special input*, that is, x or y is ± 0, $\pm\infty$, qNaN, or sNaN. For each possible case the standard [1] requires that a special value be returned. These special values follow from those in Table 3 by adjoining the correct sign, using $x/y = (-1)^{s_r}|x|/|y|$. (The standard does not specify the sign of a NaN result [1, §6.3].)

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

Table 3. Special values for $|x|/|y|$.

The standard binary encoding [1, §3.4] implies in particular Table 4. This table allows to reuse `absX` and `absY`,

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

Table 4. Floating-point data encoded by X.

which are X and Y modulo 2^{k-1} (and are computed in Section 5.1 for $k = 32$), to filter out all special input (x, y). Indeed, x or y is in $\{\pm 0, \pm\infty, \text{qNaN}, \text{sNaN}\}$ if and only if `absX` or `absY` is in $\{0\} \cup \{2^{k-1} - 2^{p-1}, \ldots, 2^{k-1} - 1\}$, which is equivalent to

$$\max(\texttt{absXm1}, \texttt{absYm1}) \ge 2^{k-1} - 2^{p-1} - 1, \qquad (25)$$

with $\texttt{absXm1} = (\texttt{absX} - 1) \mod 2^k$ and $\texttt{absYm1} = (\texttt{absY} - 1) \mod 2^k$.

Assume now that (x, y) has been filtered out by (25). Table 3 then requires that a qNaN be returned if `absX` equals `absY`, or if $\max(\texttt{absX}, \texttt{absY}) > 2^{k-1} - 2^{p-1}$. Otherwise, if x is finite (that is, $\texttt{absX} < 2^{k-1} - 2^{p-1}$) and y is nonzero (that is, `absY` is nonzero) then ± 0 is returned, else $\pm\infty$ is returned. Hence the code below when $(k, p) = (32, 24)$:

```
absXm1 = absX - 1;              absYm1 = absY - 1;
if( max(absXm1,absYm1) >= 0x7F7FFFFF ) {
   Inf = Sr | 0x7F800000;   Max = max(absX,absY);
   if( absX == absY || Max > 0x7F800000 )
      return Inf | 0x00400000 | Max;      // qNaN
   if( (absX < 0x7F800000) && absY ) return Sr;
   return Inf;
}
```

Here `absXm1` and `absYm1` are `unsigned int`, so that addition is done modulo 2^{32}. Note also that thanks to the use of the `Max` variable the payload of the qNaN result is one the input payloads, as recommended in [1, §6.2.3].

6. Experimental results

We report here some experiments done with the complete division code that follows immediately from the C codes of Sections 4 and 5.

Validation of the complete division code. Although our code has not been tested exhaustively, it has been validated using two classical test programs for IEEE binary floating-point arithmetic, and especially for division: the *Extremal Rounding Tests Set* [13] and the *TestFloat* package [7]. This was done by compiling the code with Gcc under Linux and by using the software implementations of `max` and `mul` given in Appendix A.

Results obtained with the ST200 VLIW compiler. The same code has then been compiled with the ST200 VLIW compiler (version 2-0-0-A 20061215) based on the Open64 compiler technology, re-targeted to the ST200 family.

This compiler can produce an annotated assembly output, with specific scheduling annotations giving the date at which each bundle (a group of up to 4 instructions) is scheduled. In addition, one peculiarity of the ST200 architecture is that it provides a partial predication support in the form of conditional selection instructions, that are used by the compiler [2] to generate branch-free code for all the subroutines described in this article.

Using optimization level -O3, the assembly produced in our case indeed consists of fully if-converted straight-line

code in which the number of instructions as well as the latency is the same regardless of the nature of the input (normal numbers or special values). The table below collects these two values together with the number of instructions per cycle (IPC) and the code size:

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

Since the ST231 has four issues, the IPC value indicates clearly the parallel nature of our approach.

Also, since one cycle is necessary for the last two instructions to select and return the result, the other 85 instructions must have been scheduled on the 4 issues during the first 26 cycles. This value of 26 cycles is close to the ideal value of $22 = \lceil 85/4 \rceil$.

Finally, in the same context and for the same problem (that is, software implementation of binary32 floating-point division, correctly-rounded to nearest even and without subnormal numbers), the previously fastest implementation had a latency of 48 cycles [19, p. 104]. Thus, the speed-up brought by our approach is by a factor of 1.78.

7. Conclusion and future work

In this paper, we have focused on the binary32 format and on a particular 32-bit architecture. However, the approach we have presented in Sections 4 and 5 for generating/validating polynomial evaluation codes and for designing complete division codes, should in principle be usable for other values of the key parameters k, p, and $e_{\max}$. This could be of interest when optimizing division codes for other processors and/or other floating-point formats (like binary64, binary128, or smaller formats used in computer graphics).

This work also suggests two research directions, which we are currently investigating. First, gradual underflow is important [9] and our division code should definitely be extended in order to support subnormal numbers while keeping the latency as low as possible; similarly, all the rounding attributes prescribed in [1] should be implemented. Second, in some important special cases such as inversion (instead of division) or faithfully rounded results (instead of correctly rounded results), further optimizations should be easy to implement and validate by using the generation/validation tools and the design method that we have presented here.

Acknowledgements

This work was supported by "Pôle de compétitivité mondial" Minalogic and by the ANR project EVA-Flo.

References

[1] IEEE standard for floating-point arithmetic. IEEE Std. 754-2008, pages 1–58, Aug. 2008.

[2] C. Bruel. If-conversion SSA framework for partially predicated VLIW architectures. In *Digest of the 4th Workshop on Optimizations for DSP and Embedded Systems*, 2006.

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

[4] J.-A. P. D. Piso and J. D. Bruguera. Analysis of the impact of different methods for division/square root computation in the performance of a superscalar microprocessor. *Journal of Systems Architecture*, 49(12-15):543–555, 2003.

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

[6] J. Harrison, T. Kubaska, S. Story, and P. T. P. Tang. The computation of transcendental functions on the IA-64 architecture. *Intel Technology Journal*, 1999.

[7] J. Hauser. The SoftFloat and TestFloat Packages. Available at `http://www.jhauser.us/arithmetic/`.

[8] C.-P. Jeannerod, H. Knochel, C. Monat, and G. Revy. Computing floating-point square roots via bivariate polynomial evaluation. Technical Report RR2008-38, LIP, Oct. 2008.

[9] W. Kahan. A brief tutorial on gradual underflow. Available at `http://www.cs.berkeley.edu/~wkahan/ARITH_17U.pdf`, July 2005.

[10] C. Q. Lauter. *Arrondi correct de fonctions mathématiques - fonctions univariées et bivariées, certification et automatisation*. PhD thesis, ÉNS Lyon, France, 2008.

[11] P. W. Markstein. Computation of elementary functions on the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1):111–119, 1990.

[12] P. W. Markstein. Software division and square root using Goldschmidt's algorithms. In *Proc. of the 6th Conference on Real Numbers and Computers*, pages 146–157, 2004.

[13] D. W. Matula and L. D. McFearin. Extremal rounding test sets. Available at `http://engr.smu.edu/~matula/extremal.html`.

[14] G. Melquiond. *De l'arithmétique d'intervalles à la certification de programmes*. PhD thesis, ÉNS Lyon, France, 2006.

[15] S. F. Oberman and M. J. Flynn. Fast IEEE rounding for division by functional iteration. Technical Report CSL-TR-96-700, Computer Systems Laboratory, Dept. on Electrical Engineering and Computer Science, Stanford University, 1996.

[16] S. F. Oberman and M. J. Flynn. Design issues in division and other floating-point operations. *IEEE Trans. Comp.*, 46:154–161, 1997.

[17] S. F. Oberman and M. J. Flynn. Division algorithms and implementations. *IEEE Trans. Comp.*, 46(8):833–854, 1997.

[18] J.-A. Piñeiro and J. D. Bruguera. High-speed double-precision computation of reciprocal, division, square root and inverse square root. *IEEE Trans. Comp.*, 51(12):1377–1388, 2002.

[19] S.-K. Raina. *FLIP: a Floating-point Library for Integer Processors*. PhD thesis, ÉNS Lyon, France, 2006.

[20] E. M. Schwarz. Rounding for quadratically converging algorithms for division and square root. In *Proc. of the 29th Asilomar Conference on Signals, Systems and Computers*, pages 600–603. IEEE Computer Society, 1995.

A. Implementing the `max` and `mul` instructions

A C99 implementation of `max` and `mul` is as follows:

```
static inline
unsigned int max(unsigned int A, unsigned int B)
{ return A > B ? A : B; }
```

```
static inline
unsigned int mul(unsigned int A, unsigned int B)
{
  unsigned long long int t0 = A;
  unsigned long long int t1 = B;
  unsigned long long int t2 = ( t0 * t1 ) >> 32;
  return t2;
}
```

The ST200 compiler is able to generate a single instruction when compiling the `max` or the `mul` function: it generates respectively the `maxu` and `mul64hu` ST200 instructions. This is interesting, since there is no need to write code using specific instrinsic functions, while reaching best efficiency.

B. Exact values of error bounds

Approximation error / Rounding error bounds

$$\theta_1 = \frac{3266622421341058727961746075004097830 2345}{2^{162}}$$

$$\eta_1 = \frac{9135129223254018322301121960905175404008375 2329}{2^{183}}$$

$$\theta_2 = \frac{3266610365594806277172776243763743946 6801}{2^{162}}$$

$$\eta_2 = \frac{9135230354171421854756629810036826674069999 9537}{2^{183}}$$

$$\theta_3 = \frac{1594904299976821437055348852066514943 0143}{2^{161}}$$

$$\eta_3 = \frac{4889745091520622332913501641066406029124798 7071}{2^{182}}$$

$$\theta_4 = \frac{1232560802107064287628542795321576594934 59}{2^{164}}$$

$$\eta_4 = \frac{4275548200828094949386040834528685521254859213 63}{2^{185}}$$

C. Useful hexadecimal constants

For convenience' sake the table below displays the decimal value and/or the bit string of each of the hexadecimal constants appearing in some code sequences of Section 4 and Section 5.

`0x80000000`	$2^{31} = (1\underbrace{00\ldots00}_{31\text{ zeros}})_2$
`0x7FFFFFFF`	$(0\underbrace{11\ldots11}_{31\text{ ones}})_2$
`0xFF`	$(\underbrace{00\ldots00}_{24\text{ zeros}}\underbrace{11111111}_{8\text{ ones}})_2$
`0xFFFFFFC0`	$(\underbrace{11\ldots11}_{26\text{ ones}}\underbrace{000000}_{6\text{ zeros}})_2$
`0x40`	2^6
`0x7F800000`	$2^{31} - 2^{23} = (0\underbrace{11111111}_{8\text{ ones}}\underbrace{00\ldots00}_{23\text{ zeros}})_2$
`0xFFFFFF00`	$2^{32} - 2^8 = (\underbrace{11\ldots11}_{24\text{ ones}}\underbrace{00000000}_{8\text{ zeros}})_2$
`0x7F7FFFFF`	$2^{31} - 2^{23} - 1$
`0x00400000`	$(\underbrace{00\ldots00}_{9\text{ zeros}}1\underbrace{00\ldots00}_{22\text{ zeros}})_2$