Division by Constant for the ST100 DSP Microprocessor

Jean-Michel Muller and Arnaud Tisserand
Arénaire project (CNRS–ENS Lyon–INRIA–UCBL)
LIP, ENS Lyon. 46 allée d'Italie
F–69364 Lyon Cedex 07, France
E-mail: {firstame.lastname}@ens-lyon.fr

Benoît de Dinechin and Christophe Monat
STMicroelectronics
12 rue Jules Horowitz
F–38019 Grenoble, France
Email: {firstame.lastname}@st.com

Abstract

Algorithms for Euclidean (i.e., integer) division by a constant operation are presented. They allow fast computation for some values of the divisor (known at compile time) or also when both quotient and modulus are required. These algorithms are based on the multiply-accumulate instruction and the 40-bit arithmetic available in DSPs such as the ST100 DSP from STMicroelectronics. The results are demonstrated in the case of standard speech coding applications.

1. Introduction

Significant optimizations of the speed, memory requirement and power consumption of signal processing applications can be achieved by using dedicated arithmetic operations instead of general ones whenever possible. In this paper we focus on algorithms for DSP (*digital signal processor*) implementation of an efficient Euclidean (i.e., integer) division by a constant.

Instead of using rather slow algorithms for "general" division, when the divisor is known *a priori*, it is possible to generate specialized division by a constant code on-the-fly (i.e., at compile time). Advantages expected are not only reduced execution time, but also the facilitation of other optimizations that are inhibited by the call sequence. Let us now give a few examples of possible applications of an efficient division by a constant algorithm. Most of them are interesting for designing efficient compilers.

- *Trip count computations:*

  Some DSPs provide advanced hardware support to speed-up execution of loops: *hardware loops*. Many well-formed loops can be transformed at compile time into a finite loop whose trip count must be computed at run time. For instance, consider the following fragment of C code:

```
for ( i=i0 ; i<imax ; i+=STEP )
   /* loop code */
```

  The upper bound on the number of times the loop will be executed is

$$\left\lceil \frac{\texttt{imax - i0}}{\texttt{STEP}} \right\rceil = \left\lfloor \frac{\texttt{imax - i0 + STEP - 1}}{\texttt{STEP}} \right\rfloor$$

  Since `STEP` is most often known at compile time, this run time computation can be partly reduced by using a division by constant method.

- *Addresses of objects difference:*

  In ANSI C/C++, computing `p-q` where `p` and `q` are of type `T*` requires generating code that evaluates `(p-q)/sizeof(T)`.

  Some critical library codes use this feature (and therefore generate costly division calls), and could benefit from the availability of a division by a constant algorithm. It is worth noticing that in such a case, *we know that the remainder of the division is always zero*. We

Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH'05)
1063-6889/05 $20.00 © 2005 **IEEE**

can try to take advantage from that to design an even faster division algorithm.

- *Hash-tables:*

  As pointed out by Mosberger [8], computing a hash-table index typically involves dividing by a prime integer constant. Mosberger cites an example (the time it takes to look-up all symbols in the standard C library), for which replacing "general" division by a more specific algorithm leads to a 62% improvement.

- *Radix conversions:*

  Radix-10 to radix-2 as well as radix-2 to radix-10 conversions are frequently needed when reading or displaying/storing variables. These conversions do not necessarily need to be fast (the time of computation is of course much smaller than the delay required to write or read in a file), but they must be correct and the code size should be as small as possible. Such conversions are best performed using integer division by 10 (quotient and modulus are required).

Compared to previous solutions, our algorithms allow to get the quotient as well as the modulus (or remainder) very quickly. They rely on the multiply-accumulate (MAC) operation and the 40-bit arithmetic provided in many DSPs.

This paper is organized as follows. The arithmetic resources of the DSP used in this work, the ST100 from STMicroelectronics, are presented in Section 2. Section 3 presents the definitions and notations used below. Previous work is presented in Section 4. Our algorithms are presented in Section 5. Section 6 briefly presents the application of our division by a constant algorithms on a speech coding algorithms on the ST100 processor.

2. The ST100 DSP from STMicroelectronics

STMicroelectronics's ST100 DSP core architecture [9] is aimed at advanced embedded system-on-chip (SoC) solutions for wireless terminals and base stations, networking, broadband modems, voice-over-IP, data storage and mobile multimedia applications where high performance, low power and fast development are all essential. The first implementation, called ST120, includes 5 processing units (2 for data arithmetic), 16 working data registers and 16 control registers, giving 3200 MOP/s and 800 MMAC/s (Mega MAC per second) at 400 MHz in a 1.2 V static design.

All data registers are 40-bit wide and can be used with signed/unsigned, $16, 32$ or 40 integers or fractional values (with several formats). For multiplication (and MAC), as the 2 operands of the multiplier are only 16-bit wide, both the lower or the upper 16-bit half words (of the 32-bit lower bits in the data register) can be considered. The ST100 processor is able to execute 2 arithmetic instructions per clock cycle. Those instructions can be 40-bit ALU (*arithmetic and logic unit*) operations or $16 \times 16 \pm 40 \rightarrow 40$ MAC operations. The ST100 core provides 2 ALUs and 2 multiply-and-accumulate (or subtract) units. The full 40-bit data registers and data-path allow up to 255 consecutive $16 \times 16 \pm 32 \rightarrow 40$ multiply-accumulate operations without any overflow.

3. Definitions and Notations

Division by a constant is also dealt with in [5], but in a totally different context. The authors of that paper wish to implement *floating-point* division, assuming that a full-precision floating-point fused multiply-add unit is available. Here, we are interested in *integer* division, and we assume we use the integer units of the ST100 processor.

An important point is to clearly define what we mean by "integer division". As pointed out by Boute [4], the definitions of functions `div` and `mod` in the computer science literature and in programming languages sometimes differ. As a consequence, when a and b are not both positive, the result returned when computing $a \bmod b$ and a div b may differ, depending on the programming environment (and may be different from what the programmer had in mind). This of course may be a severe source of bugs and problems of portability.

The definition we chose in this paper is the following:

Definition 1 *The quotient Q and remainder R of the division of A by B are defined by :*

$$\begin{cases} A = BQ + R \\ 0 \leq |R| < |B| \\ R \text{ has the same sign as } A \end{cases}$$

This convention is not the best one from a mathematical point of view (the best one is probably the one defined from Euclide's theorem [4]), and yet it is used in most C compilers. Anyway, adapting our algorithms to other definitions is rather straightforward. Using this definition, and denoting respectively $Q(A, B)$ and $R(A, B)$ the quotient and remainder of the division of A by B, we get:

- $Q(35, 4) = 8$ and $R(35, 4) = 3$;
- $Q(35, -4) = -8$ and $R(35, -4) = 3$;
- $Q(-35, 4) = -8$ and $R(-35, 4) = -3$;
- $Q(-35, -4) = +8$ and $R(-35, -4) = -3$.

The following relations allow to easily deduce the general case from the case of unsigned integers:

- $Q(A,-B) = -Q(A,B)$ and $R(A,-B) = R(A,B)$;
- $Q(-A,B) = -Q(A,B)$ and $R(-A,B) = -R(A,B)$.

When implementing arithmetic operators, it is essential to make sure that the potential overflow problems are addressed correctly. Concerning division with the above chosen convention, the overflow detection is easily done: if A and B are representable on a p-bit binary format (i.e., they are between 0 and $2^p - 1$ if we assume an unsigned integer format, or they are between -2^{p-1} and $2^{p-1} - 1$ if we assume 2's complement representation) then $Q(A,B)$ and $R(A,B)$ are always representable in the same format *unless* we are in $2's$ complement *and* $A = -2^{p-1}$ *and* $B = -1$. This only exception is easily handled (the best solution is to include it in a separate treatment of the otherwise obvious case $B = -1$). *In all that follows, we now assume that the straightforward cases $B = 0, \pm 1, \pm 2^k$ are handled separately, and we focus on the other cases.*

In the sequel of the paper, we wish to divide A by B, where A and B are 32-bit integers, and B is known at compile time. Without loss of generality, we assume that B is nonnegative. Since we assume that B is not a power of 2, we have $B \geq 3$. When fast multipliers are not available, one can perform division by a constant as a sequence of shifts and additions [3]. That was frequent with old circuits, and that still can be used, for instance if one wishes to do some computations in the address unit of the ST100 (in this work, we design algorithms for the data unit of the ST100).

Now, most techniques [7, 6], including ours, consists in multiplying A by some approximation to $1/B$ and then in performing some correcting step, to get an exact quotient and remainder. The approximation must be accurate enough to make the correcting step simple, and yet must be so that the initial multiplication be as simple as possible. The correcting step is very architecture-dependent. One of the main issues is to avoid tests as much as possible.

4. Granlund and Montgomery's algorithm

Granlund and Montgomery's algorithm [6] is based on the following result.

Theorem 1 *Suppose N, m, ℓ, B are positive integers such that*

$$2^{N+\ell} \leq m \times B \leq 2^{N+\ell} + 2^{\ell}$$

then $\lfloor A/B \rfloor = \lfloor m \times A/2^{N+\ell} \rfloor$ for every integer A with $0 \leq A < 2^N$.

Hence, when A (unknown at compile time) and B (known at compile time) are N-bit numbers, division by B is replaced by multiplication by a suitable value m. Since m can be as large as 2^{N+1}, Granlund and Montgomery replace this multiplication by a shift, an addition and an N-bit $\times N$-bit multiplication:

$$A \times m = A \times 2^N + A \times (m - 2^N).$$

On the ST100 DSP, this will require 4 $16 \times 16 + 40$-bit multiplications-accumulations, an addition, and shifts. Now our goal is to check whether, for special cases, we can get a faster and/or smaller code.

5. Division of unsigned integers

We assume that A is a positive 32-bit integer. Signed values are handled using the formulas given in Section 3. Our goal is to get a simple algorithm in some cases that, in practice, occur quite often (e.g., we know in advance that the remainder is zero, or the divisor is small).

We will use the following, straightforward, lemma.

Lemma 1 *Let A and B be nonnegative integers. If ρ is an approximation to A/B such that*

$$0 \leq \rho - \frac{A}{B} < \frac{1}{B},$$

then

$$\lfloor \rho \rfloor = \left\lfloor \frac{A}{B} \right\rfloor$$

This case is very frequent, since is corresponds to the calculation of addresses of objects difference (see Section 1). Assume we divide A by B, where A and B are 32-bit unsigned integers, B is known at compile time, and A is a multiple of B. Define

$$\begin{aligned} A &= A_H 2^{16} + A_L \\ B &= B_H 2^{16} + B_L \end{aligned}$$

Assuming $B \geq 3$, one can compute at compile time two integers Z_H and Z_L such that

$$Z = \frac{1}{B} = Z_H 2^{-17} - Z_L 2^{-33} - Z_R 2^{-33}$$

where

$$\begin{cases} Z_H \neq 0 & \text{is a 16-bit integer} \\ Z_L & \text{is a 16-bit integer} \\ 0 \leq Z_R < 1 \end{cases}$$

At run-time, we compute

$$\begin{aligned} \rho_1 &= A_H Z_H \\ \rho_2 &= -A_H Z_L + A_L Z_H \\ \hat{Q} &= \left\lfloor 2^{-1}\rho_1 + 2^{-17}\rho_2 \right\rfloor \end{aligned}$$

(i.e., $\hat{Q}$ is obtained by first computing $\rho_1+2^{-16}\rho_2$, and then by shifting the result by one position to the right.)

Define $\rho = 2^{-1}\rho_1 + 2^{-17}\rho_2$. we have:

$$\begin{array}{rcl} \rho - \frac{A}{B} & = & \rho - AZ = (A_L Z_L + AZ_R)2^{-33} \\ & \leq & \left((2^{16}-1)Z_L + (2^{32}-1)Z_R\right)2^{-33} \end{array}$$

This last bound is always less than 1. Hence, since $AZ = A/B$ is an integer , and $0 \leq \rho - AZ < 1$, we necessarily have $\lfloor \rho \rfloor = AZ$, therefore $\hat{Q}$ is the quotient we are looking for.

This case is easily handled. Let $Z = 1/B$, with $B > 2$. At compile time, we compute two 16-bit positive integers Z_H and Z_L that satisfy

$$Z = Z_H \times 2^{-17} + (Z_L - Z_R) \times 2^{-33},$$

where $0 \leq Z_R < 1$ (hence, $Z_H \times 2^{-17} + Z_L \times 2^{-33}$ overestimates $1/B$).

At run-time, we compute

$$\left\{ \begin{array}{ll} \rho_1 & = AZ_H \\ \rho_2 & = AZ_L \\ \hat{Q} = \lfloor 2^{-17}\rho_1 + 2^{-33}\rho_2 \rfloor & \end{array} \right.$$

($\hat{Q}$ is obtained by first computing $\rho_1 + 2^{-16}\rho_2$, and then by shifting the result by 17 positions to the right)

We easily get:

$$0 \leq \left(2^{-17}\rho_1 + 2^{-33}\rho_2\right) - AZ < (2^{16}-1)\times 2^{-33} < 2^{-17} < \frac{1}{B},$$

Therefore, from Lemma (1), $\hat{Q} = 1/B$. If the remainder is required, it is directly obtained by $R = A - B\hat{Q}$, without any risk of overflow.

We start from:

$$\begin{array}{rcl} A & = & A_H 2^{16} + A_L \\ B & = & B_H 2^{16} + B_L \end{array}$$

At compile time, we compute two positive integers Z_H and Z_L such that

$$Z = \frac{1}{B} = Z_H 2^{-16-\delta} - Z_L 2^{-32-\delta} - Z_R 2^{-32-\delta}$$

where

$$\left\{ \begin{array}{ll} Z_H \neq 0 & \text{is a 16-bit integer} \\ Z_L & \text{is a 16-bit integer} \\ 0 \leq Z_R < 1 & \\ \delta & \text{is an integer, as large as possible} \end{array} \right.$$

This computation of Z_H et Z_L can be done as follows:

$power \longleftarrow 65536$
$\delta \longleftarrow 0$
$Z_H \longleftarrow power\ DIV\ B$
while $Z_H \leq 32767$ **do**
  $\delta := \delta + 1$
  $power := 2 \times power$
  $Z_H := power\ DIV\ B$
end
$Z_H := Z_H + 1$
$rem := (Z_H/power - 1/B) * 2^{(32+\delta)}$
$Z_L := trunc(rem)$
$Z_R := rem - Z_L$

At run-time, we compute:

$$\begin{array}{rcl} \rho_1 & = & A_H Z_H \\ \rho_2 & = & -A_H Z_L + A_L Z_H \\ \hat{Q} & = & \lfloor 2^{-\delta}\rho_1 + 2^{-16-\delta}\rho_2 \rfloor \end{array}$$

(i.e., $\hat{Q}$ is obtained by first computing $\rho_1+2^{-16}\rho_2$, and then by shifting the result by δ positions to the right.)

Define $\rho = 2^{-\delta}\rho_1 + 2^{-16-\delta}\rho_2$. We have:

$$\begin{array}{rcl} \rho - \frac{A}{B} & = & \rho - AZ = (A_L Z_L + AZ_R)2^{-32-\delta} \\ & \leq & \left((2^{16}-1)Z_L + (2^{32}-1)Z_R\right)2^{-32-\delta} \end{array}$$

At compile time, we can compute (for a given value of B) the bound:

$$\left((2^{16}-1)Z_L + (2^{32}-1)Z_R\right)2^{-32-\delta}$$

When that bound is less than Z (which happens 231 times for the first 1000 values of B, and 16185 times for the first 65535 values), the required quotient is $\hat{Q}$, from Lemma (1).

The values of B less than 200 for which this special case occurs are: 3, 5, 6, 10, 11, 12, 13, 17, 20, 22, 24, 26, 29, 34, 40, 43, 44, 48, 52, 58, 59, 67, 68, 80, 81, 85, 86, 88, 96, 104, 116, 118, 129, 134, 136, 137, 139, 141, 143, 149, 157, 160, 162, 163, 169, 170, 172, 175, 176, 181, 187, 192 and 199.

Example

Assume $B = 85$. The binary representation of Z is $0.000000110000011000000110\ldots$. We have:

$$\left\{ \begin{array}{rcl} \delta & = & 6 \\ Z_H & = & 49345_{10} \\ & = & 1100000011000001_2 \\ Z_L & = & 16191_{10} \\ & = & 0011111100111111_2 \\ Z_R & = & 0.2470588\cdots_{10} \\ & = & 0.00111111001111110\cdots_2 \end{array} \right.$$

For $A = 546559$, we get $A_H = 8$ and $A_L = 22271$. We find $\rho_1 = 394760$ and $\rho_2 = 1098832967$, which gives $\hat{Q} = 6430$, which is the right quotient.

When the simplifications presented in Sections 5.1, 5.2, and 5.3 do not apply, we need a slightly more complex algorithm. Let

$$\left\{\begin{array}{rcl} A & = & A_H \times 2^{16} + A_L \\ B & = & B_H \times 2^{16} + B_L \\ Z & = & 1/B \\ & = & Z_H \times 2^{-17} + (Z_L + Z_R) \times 2^{-33} \end{array}\right.$$

where

- A_H, A_L, B_H, B_L, Z_H et Z_L are integers between 0 and $2^{16} - 1$;
- the first fractional bit of Z is zero;
- Z_R is a real number between 0 and 1.

We get :

$$0 \leq Z - (Z_H \times 2^{-17} + Z_L \times 2^{-33}) < +2^{-33}.$$

At run-time, we successively compute:

$$\left\{\begin{array}{rcl} \rho_1 & = & \frac{1}{2} A_H Z_H \\ \rho_2 & = & (A_H Z_L + A_L Z_H) \times 2^{-17} \end{array}\right.$$

It is worth noticing that these calculations are error-free and overflow-free. Also, as soon as $B > 2^{17}$, $Z_H = 0$. Since this is known at compile time, the computation of ρ_1 and ρ_2 is simplified (in that case, $\rho_1 = 0$ and $\rho_2 = A_H Z_L \times 2^{-17}$). Define $\hat{Q}$ as the integer part of $\rho_1 + \rho_2$ (obtained by a mere right shift of the 40-bit register that contains $\rho_1 + \rho_2$).

We have:

$$AZ - (\rho_1 + \rho_2) = (A_L Z_L + A Z_R) \times 2^{-33} \qquad (1)$$

This gives:

$$0 \leq AZ - (\rho_1 + \rho_2)$$
$$\leq \left[(2^{16} - 1)^2 + (2^{32} - 1)\right] \times 2^{-33} = 1 - 2^{-17}$$

Now define Q as the exact required quotient of the division of A by B.

From:

$$\left\{\begin{array}{rcll} Q & = & \lfloor A/B \rfloor & = \lfloor AZ \rfloor \\ \hat{Q} & = & \lfloor \rho_1 + \rho_2 \rfloor & \\ 0 & \leq & AZ - (\rho_1 + \rho_2) \leq 1 - 2^{-17} < 1 & \end{array}\right.$$

We deduce that Q is equal to $\hat{Q}$ or to $\hat{Q} + 1$. What follows consists in trying to evaluate the "tentative remainder" $A - B(\hat{Q}+1)$. Since $-B \leq A - B(\hat{Q}+1) < +B$, that tentative remainder is representable without overflow. Also, the largest possible value of $B(\hat{Q} + 1)$ is $|A + B|$ which is representable with the 40-bit format (since A and B are 32-bit integers), so this multiplication cannot lead to an overflow.

Define $\bar{Q}_L$ and $\bar{Q}_H$ as $\hat{Q}+1 = \bar{Q}_H \times 2^{16} + \bar{Q}_L$. Computing the product $B(\hat{Q} + 1)$ does not require the calculation of all partial products:

- if $B_H = 0$ then $B_H \bar{Q}_H = 0$;
- if $B_H \neq 0$ then either B equals 2^{16} or $2^{16} + 1$ (in such cases, one may have $\bar{Q}_H = 1$ but the product $B(\hat{Q}+1)$ is straightforwardly computed), or $B \geq 2^{16}+2$. In that last case, $\bar{Q}_H = 0$, therefore $B_H \bar{Q}_H = 0$.

Hence, in almost all cases, $B_H \bar{Q}_H = 0$. Therefore, in the calculation of

$$\hat{R} = A - B(\hat{Q}+1) = A - (B_H \bar{Q}_L + B_L \bar{Q}_H) \times 2^{16} - B_L Q_L,$$

at least one term (either $B_H \bar{Q}_L$ or $B_L \bar{Q}_H$), is known in advance to be zero. The quotient Q and the remainder R of the Euclidean division are obtained as:

- if $\hat{R} < 0$, then $Q = \hat{Q}$ and $R = \hat{R} + B$;
- otherwise, $Q = \hat{Q} + 1$ and $R = \hat{R}$.

Our calculation leads to adding to the 3 multiplications-additions needed to compute ρ_1 and ρ_2 the computation of a tentative remainder. Hence, if the remainder is not needed, it is preferable to use Montgomery and Grandlung's division algorithm. If we need to compute the quotient *and* remainder, our algorithm is preferable.

Example

Assume $B = 5604_{10} = 15E4_{16}$. We find:

- $Z = 0.00178443968593861527480371163454 67\ldots_{10}$
- $Z_H = 23_{10} = 17_{16}$
- $Z_L = 25494_{10} = 6396_{16}$
- $Z_R = 0.01855817273376159885\ldots_{10}$
- $B_H = 0$
- $B_L = 5604_{10} = 15E4_{16}$

If $A = 932729_{10} = E3B79_{16}$ then:

- $\rho_1 = 161_{10} = A1_{16}$
- $\rho_2 = 707091_{10} \times 2^{-17} = 5.65098_{16}$
- $\hat{Q} = \lfloor \rho_1 + \rho_2 \rfloor = 166_{10} = A6_{16}$

Therefore $\bar{Q}_H = 0$, $\bar{Q}_L = 167_{10} = A7_{16}$. Since $B_H = 0$, $\hat{R}$ is computed as:

$$\hat{R} = A - B(\hat{Q} + 1) = A - B_L \bar{Q}_H \times 2^{16} - B_L Q_L.$$

We then get: $\hat{R} = -3139$. Therefore:

- $Q = \hat{Q} = 166_{10} = A6_{16}$;
- $R = \hat{R} + B = 2465_{10} = 9A1_{16}$.

6. Application

To demonstrate the benefits of our division by a constant methods, we use the ETSI (European Telecommunication Standards Institute) and ITU (International Telecommunication Union) reference implementations of speech coding algorithms such as the ETSI EFR-5.1.0, the ETSI AMR-NB [1], the ITU G.723.1 and G729 [2]. Our division by a constant algorithms have been implemented in the C/C++ compiler from STMicroelectronics for the ST100 processor. The benchmarks have been compiled with the original division algorithm and with our division by a constant algorithms.

It should be noted that the original run-time division support of the ST100 core is already quite efficient, despite being a software only (library) implementation: this algorithm runs in time proportional to the difference of absolute magnitudes between the dividend and the divisor, and also takes advantage of specific ST100 instructions: the Leading Zero Count (LZC) to compute the difference of magnitudes between the dividend and the divisor, and the Viterbi instruction to implement the compare and subtract of the inner loop in a single step.

The benefits of this replacement are not limited to the sole net gain due to the smaller sequence, as they also enable other optimizations:

- Elimination of all edge cases such as trivial division or division by zero.
- Call elimination removes optimization barriers, for instance exposing another level of hardware loop and freeing more registers that should be otherwise saved.
- Arithmetic operations can easily be speculated (in a model where we accept to ignore sticky effects, which is always the case with non-DSP arithmetic), thus giving more scheduling freedom.
- Partial redundancies can be exposed, thus eliminated by further optimizations.

A drawback is a slight amount of code size expansion but this effect is sometimes balanced by better opportunities to use the machine resources, filling otherwise unused scheduling slots.

Performance of the vocoder algorithm families is usually measured in MCycles/s, defined as the frequency at which the core must run to sustain real-time speech coding and decoding. Since for a given compression rate speech is treated as fixed duration samples, the performance computation is straightforward.

This reduction of MCycles/s should be balanced against the possible increase in code size, that has a negative impact on the final circuit cost. Thus we measure the code bloat due to inline expansion, that we define as :

$$\text{Bloat} = \frac{\text{Code size using optimization}}{\text{Code size without the optimization}}$$

Table 1 presents the results for several benchmarks. For each test, the number of executed cycles, the code size, the speedup (original *vs*. using our algorithms) and the code bloat are reported.

7. Conclusion

We developed and implemented a new integer division by a constant method, that takes advantage of the native 40-bit and MAC arithmetic capabilities found on modern digital signal processors such as the STMicroelectronics ST100 family.

Our algorithms allow to speedup the computation for some special values of the divisor (known at compile time), for reduced format operands, and when the remainder is known to be zero. When both the quotient and the modulus are required, our algorithms ensure faster computations. All those cases can be implemented in a compiler to chose, on-the-fly, the best possible algorithm for each value of the divisor.

Our algorithms have been implemented in the ST100 compilation tools and have been validated on standard speech coding algorithms from the ITU (International Telecommunication Union) [2] and the ETSI (European Telecommunication Standards Institute) [1].

References

[1] European telecommunications standards institute – ETSI: GSM technical activity, SMG11 (speech) working group. available at `http://www.etsi.org`.

[2] International telecommunication union (ITU). available at `http://www.itu.int`.

[3] E. Artzy, J. Hinds, and H. Saal. A fast division technique for constant divisors. *Communications of the ACM*, 19(2):98–101, Feb. 1976.

[4] R. T. Boute. The euclidean definition of the functions div and mod. *ACM Trans. Program. Lang. Syst.*, 14(2):127–144, 1992.

[5] N. Brisebarre, J.-M. Muller, and S. Raina. Accelerating correctly rounded floating-point division when the divisor is known in advance. *IEEE Transactions on Computers*, 53(8):1069–1072, Aug. 2004.

[6] T. Granlund and P. L. Montgomery. Division by invariant integers using multiplication. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 61–72. ACM Press , New York, NY, USA, Aug. 1994.

Bench/Function/Variant (Divisor)	Original division Cycles (Bytes)	Our division Cycles (Bytes)	Improvement Speedup (Bloat)
EFR/c1035pf_set_sign/C (5)	1554 (484)	1220 (512)	1.27 (1.17)
EFR/c1035pf_set_sign/LAI (5)	1652 (472)	1323 (504)	1.25 (1.07)
EFR/c1035pf_cor_h_x/C (5)	3496 (388)	3175 (416)	1.10 (1.07)
EFR/c1035pf_cor_h_x/LAI (5)	2553 (336)	2194 (352)	1.16 (1.05)
EFR/c1035pf_search_10i40/C (5)	41620 (5068)	30261 (5460)	1.38 (1.11)
EFR/c1035pf_search_10i40/LAI(5)	36490 (4720)	21876 (4832)	1.67 (1.02)

[7] D. Magenheimer, L. Peters, K. Pettis, and D. Zuras. Integer multiplication and division on the HP precision architecture. *IEEE Transactions on Computers*, 37(8):980–990, Aug. 1988.

[8] D. Mosberger. Linux and the Alpha, how to make your application fly, part 2. *Linux Journal*, 1997.

[9] STMicroelectronics. ST 122 DSP core overview handbook. available at `http://us.st.com/stonline/prodpres/dedicate/st100/document/document.htm%`, 2003.