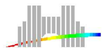
Division by Constant for the ST100 DSP Microprocessor

Jean-Michel Muller, Arnaud Tisserand (LIP)
Benoît de Dinechin, Christophe Monat (ST)



Contents

1. Context
   → *division by constant in a compiler*
2. ST100 DSP Microprocessor Core
   → *general features and arithmetic units*
3. State of the Art Algorithm
   → *definitions and Granlund and Montgomery's algorithm*
4. Our Division by Constant Algorithms for the ST100 Core
   → *optimizations and general case*
5. Application Results
   → *benchmarks: speech coding algorithms*

Division by Constant Motivations

Context: efficient Euclidean, i.e., exact or integer division by a constant algorithm (get the quotient as well as the modulus).

Some examples of possible applications in compilers:

- Trip count computed at run time for the dedicated *hardware loops*

```
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∗ (known at compile time) requires generating code that evaluates (p−q)/**sizeof**(T).

  Optimization: the remainder of the division is always zero

- Hash-tables

  Use division by a prime integer constant.

  Example from Mosberger: look-up time symbols in the standard C library.
  "general" division → specific algo. = 62% improvement

- Radix conversions

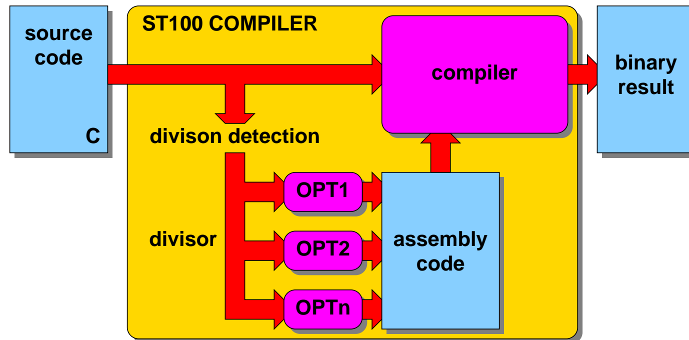
  Radix-10 ⟷ radix-2 conversions for I/O (division by 10 with quotient and modulus). Code size should be as small as possible.

Goal in the ST Compiler

Generate specialized division by a constant code on-the-fly (i.e., at compile time), when the divisor is known *a priori*.

⇝ Reduced execution time

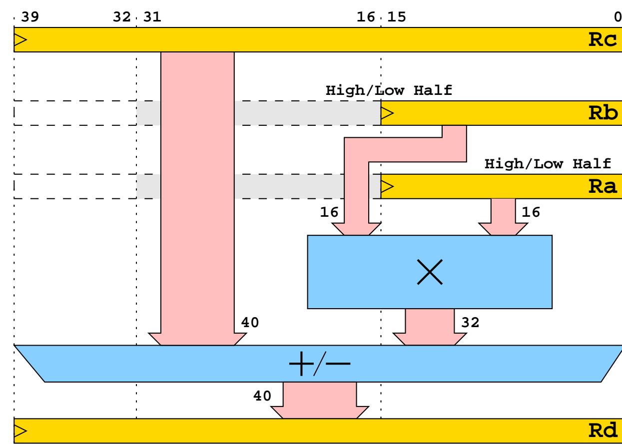
⇝ Code inlining



ST100 DSP Microprocessor Core

- Used in embedded system-on-chip solutions: *wireless terminals and base stations, networking, broadband modems, voice-over-IP, data storage and mobile multimedia applications*
- High performance, low power and fast development
- First implementation: ST120
  - 5 processing units, 32 working registers and 16 control registers
  - data registers are 40-bit wide and can be used with signed/unsigned, $16, 32$ or 40 integers or fractional values (with several formats)
  - 2 ALUs (40 bits) and 2 multiply-and-accumulate (or subtract) units
  - giving 3200 MOP/s and 800 MMAC/s (Mega MAC per second) at 400 MHz in a 1.2 V static design

ST100 Multiply and Add Unit ($16 \times 16 \pm 40 \rightarrow 40$)



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.

What is "integer division" ?

As pointed out by Boute, the definitions of the functions `div` and `mod` in the computer science literature and in programming languages differ → can be a severe source of bugs and problems of portability.

The definition we chose 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, and yet it is used in most C compilers.

Potential overflow problems ?

- A and B: representable on a p-bit binary format (unsigned integer format, or 2's complement);
- $Q(A,B)$ 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.

We now assume that the straightforward cases $B = 0, \pm 1, \pm 2^k$ are handled separately, and we focus on the other cases.

Our general problem

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 $B \geq 3$.

- Most techniques, 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: accurate enough to make the correcting step simple, and yet so that the initial multiplication be as simple as possible.
- Correcting step: very architecture-dependent. One of the main issues is to avoid tests as much as possible.

Granlund and Montgomery's algorithm (1994)

Theorem 1 *Suppose m, ℓ, B are nonnegative integers s.t.*

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

then

$$\left\lfloor \frac{A}{B} \right\rfloor = \left\lfloor \frac{m \times A}{2^{N+\ell}} \right\rfloor$$

for every integer A with $0 \leq A < 2^N$.

When A and B are N-bit numbers, division by B replaced by multiplication by m. m can be as large as $2^{N+1} \rightarrow$

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

On the ST-100, requires 4 16×16-bit multiplications-accumulations, an addition, and shifts.

Our goal: check whether, for *special cases*, we can get a faster and/or smaller code.

Division of unsigned integers

We assume that A is a positive 32-bit integer. Our goal: get a simple algorithm in 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:

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

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

Special case: the remainder is known to be zero (1/2)

Frequent case: calculation of addresses of objects difference.

Assume A and B are 32-bit unsigned integers 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

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

Special case: the remainder is known to be zero (2/2)

At run-time, we compute

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

(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.)

Property: $\hat{Q}$ is the quotient we are looking for.

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

Since $AZ = A/B$ is an integer, we necessarily have $\lfloor \rho \rfloor = AZ$.

Special case: division of 16-bit integers

Very often, B is a rather small integer.

Let $Z = 1/B$, with $B > 2$. At compile-time, 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, compute
$$\begin{cases} \rho_1 = A Z_H \\ \rho_2 = A Z_L \\ \hat{Q} = \lfloor 2^{-17}\rho_1 + 2^{-33}\rho_2 \rfloor \end{cases}$$

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

Property: $\hat{Q} = A/B$.

If the remainder is required, it is directly obtained by $R = A - B\hat{Q}$, without any risk of overflow.

Special case: get the quotient without any testing (1/3)

We start from:

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

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

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

Special case: get the quotient without any testing (2/3)

At run-time, we compute:

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

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

We have:

$$\rho - \frac{A}{B} \leq \left((2^{16}-1)Z_L + (2^{32}-1)Z_R\right) 2^{-32-\delta}$$

Special case: get the quotient without any testing (3/3)

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 $< Z$ (which happens 231 times for the first 1000 values of B, and 16185 times for the first 65535 values), the required quotient is $\hat{Q}$.

Values of B less than 200 for which this special case occurs:

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.

Application Results (1/3): speech coding benchmarks

Reference implementations from:

- ETSI (European Telecommunication Standards Institut)
- ITU (International Telecommunication Union)

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.

Remark: the original run-time division support of the ST100 core is already quite efficient (takes advantage of specific ST100 instructions: Leading Zero Count (LZC) and Viterbi instruction).

Application Results (2/3): benefits and drawback

Benefits:

- Speed improvement
- Elimination of all edge cases such as trivial division
- Call elimination removes optimization barriers
- Arithmetic operations can easily be speculated

Drawback:

- Slight amount of code size expansion
- Measure the code bloat due to inline expansion using:

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

Application Results (3/3): the results

For each test, the number of executed cycles, the code size, the speedup (original *vs.* using our algorithms) and the code bloat are reported.

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

Conclusion

- New integer division by a constant method (divisor known at compile time):
  - algorithms with proofs of correctness (overflow free)
  - implemented on ST100 DSP microprocessor cores
  - can be extended to similar DSPs
- Integrated in the ST production compiler
- Extensively tested (specific values, random values, real applications and benchmarks)
- Speed improvement: from 10% up to 60% on real benchmarks
- Drawback: code inlining → code bloat (up to 17%)

This is the end...

Questions ?

Contacts:

- jean-michel.muller@ens-lyon.fr
  arnaud.tisserand@ens-lyon.fr
- http://perso.ens-lyon.fr/jean-michel.muller/
  http://perso.ens-lyon.fr/arnaud.tisserand/
- LIP, ENS Lyon. 46 allee d'Italie. F-69364 Lyon cedex 07. France.

Thank you.