Table-based division by small integer constants

Florent De Dinechin, Laurent-Stéphane Didier

Table-based division by small integer constants

Florent de Dinechin, Laurent-Stéphane Didier

LIP, Université de Lyon (ENS-Lyon/CNRS/INRIA/UCBL)
46, allée d'Italie, 69364 Lyon Cedex 07
{Florent.de.Dinechin}@ens-lyon.fr
LIP6, Université Pierre etMarie Curie (UPMC, CNRS)
4 place Jussieu, 75252 Paris Cedex 05
{Laurent-Stephane.Didier}@upmc.fr

Abstract. Computing cores to be implemented on FPGAs may involve divisions by small integer constants in fixed or floating point. This article presents a family of architectures addressing this need. They are derived from a simple recurrence whose body can be implemented very efficiently as a look-up table that matches the hardware resources of the target FPGA. For instance, division of a 32-bit integer by the constant 3 may be implemented by a combinatorial circuit of 48 LUT6 on a Virtex-5. Other options are studied, including iterative implementations, and architectures based on embedded memory blocks. This technique also computes the remainder. An efficient implementation of the correctly rounded division of a floating-point constant by such a small integer is also presented.

1 Introduction

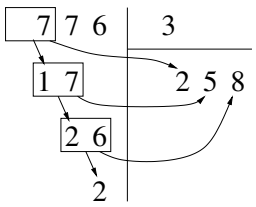
When porting applications to FPGAs, arithmetic operations should be optimized in an application-specific way whenever possible. This is the goal of the FloPoCo project [1]. This article considers division by a small integer constant, and demonstrates operators for it that are more efficient than approaches based on standard division [2] or on multiplication by the inverse [3, 4].

Division by a small integer constant is an operation that occurs often enough to justify investigating a specific operator for it. This work, for instance, was motivated by the Jacobi stencil algorithm, whose core computes the average of 3 values: this involves a division by 3. Small integer constants are quite common in such situations. Division by 5 also occurs in decimal / binary conversions. The proposed approach could also be used to interleave memory banks in numbers that are not powers of two: if we have d memory banks, an address A must be translated to address A/d in bank $A \mod d$.

Division by a constant in a hardware context has actually been studied quite extensively [3, 5, 4], with good surveys in [6, 7]. There are two main families of techniques: those based on additions and subtractions, and those based on multiplication by the inverse. In this article we introduce a technique that is, to our knowledge, new, although it is in essence a straightforward adaptation

of the paper-and-pencil division algorithm in the case of small divisors. The reason why this technique is not mentioned in the literature is probably that the core of its iteration itself computes a (smaller) division: it doesn't reduce to either additions, or multiplications. However, it is very well suited to FPGAs, whose logic is based on look-up tables (LUTs): they may implement such complex operations very efficiently, provided the size in bits of the input numbers matches the number of inputs to the hardware LUTs.

Let us introduce this technique with the help of usual decimal arithmetic. Suppose we want to divide an arbitrary number, say 776, by 3. Figure 1 describes the paper-and-pencil algorithm in this case.



We first computes the Euclidean division of 7 by 3. This gives the first digit of the quotient, here 2, and the remainder is 1. We now have to divide 176 by 3. In the second iteration, we divide 17 by 3: the second quotient digit is 5, and the remainder is 2. The third iteration divides 26 by 3: the third quotient digit is 8 and the remainder is 2, and this is also the remainder of the division of 776 by 3.

Fig. 1. Division by 3 in decimal

The key observation is that in this example, the iteration body consists in the Euclidean division of a 2-digit decimal number by 3. The first of these two digits is a remainder from previous iteration: its value is 0, 1 or 2. We may therefore implement this iteration with a look-up table that, for each value from 00 to 29, gives the quotient and the remainder of its division by 3. This small look-up table will allow us to divide by 3 numbers of arbitrary size.

In Section 2 we adapt this radix-10 algorithm to a radix that is a power of two, then chose this radix so that the look-up table matches well the fine structure of the target FPGA. We study the case of floating-point inputs in Section 3: it is possible to ensure correct rounding to the nearest for free. Section 4 provides a few results and comparisons.

2 Euclidean division of an integer by a small constant

2.1 Notations

Let d be the constant divisor, and let α be a small integer. We will use the representation of x in radix $\beta = 2^\alpha$, which may also be considered as breaking down the binary decomposition of x into k chunks of α bits (see Figure 3):

$$x = \sum_{i=0}^{k-1} x_i . 2^{-\alpha i} \quad \text{where} \quad x_i \in \{0, ..., 2^\alpha - 1\}$$

In all this section, we assume that d is not a multiple of 2, as division by 2 resumes to a constant shift which is for free in FPGAs.

2.2 Algorithm

The following algorithm computes the quotient q and the remainder r_0 of the high radix euclidean division of x by the constant d. At each step of this algorithm, the partial dividend y_i, the partial remainder r_i and one radix-2^α digit of the quotient are computed.

Algorithm 1 LUT-based computation of x/d

```
1: procedure ConstantDiv(x, d)
2:     r_k ← 0
3:     for i = k − 1 down to 0 do
4:         y_i ← x_i + 2^α r_{i+1}                    (this + is a concatenation)
5:         (q_i, r_i) ← (⌊y_i/d⌋, y_i mod d)           (read from a table)
6:     end for
7:     return q = Σ_{i=0}^{k} q_i.2^{−αi}, r_0
8: end procedure
```

Theorem 1. *Algorithm 1 computes the Euclidean division of x by d. It outputs the quotient $q = \sum_{i=0}^{k} q_i.2^{-\alpha i}$ and the remainder r_0 so that $x = q \times d + r_0$. The radix-2^α representation of the quotient q is also a binary representation, each iteration producing α bits of this quotient.*

The proof of this theorem is in appendix. The line $y_i \leftarrow x_i + 2^\alpha r_{i+1}$ is simply the concatenation of a remainder and a radix-2^α digit. Let us define γ as the size in bits of the largest possible remainder: $\gamma = \lceil \log_2(d-1) \rceil$ – this is also the size of d as d is not a power of two. Then, y_i is of size $\alpha + \gamma$ bits. The second line of the loop body, $(q_i, r_i) \leftarrow (\lfloor y_i/d \rfloor,\ y_i \mod d)$, computes a radix-$2^\alpha$ digit and a remainder: it may be implemented as a look-up table with $\alpha + \gamma$ bits of input and $\alpha + \gamma$ bits of output (Fig. 2). Here, α is a parameter which may be chosen to match the target FPGA architecture, as we show below. The algorithm computes α bits of the quotient in one iteration: the larger α, the fewer iterations are needed for a given input number size n.

The iteration may be implemented sequentially as depicted on Fig. 2, although in all the following we will focus on the fully unrolled architecture depicted on Fig. 3, which enables high-throughput pipelined implementations.

It should be noted that, for a given d, the architecture grows linearly with the input size n, where general division or multiplication architectures grow quadratically.

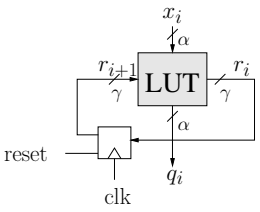


Fig. 2. LUT-based squential division by a constant of a radix-2^α digit extended by a remainder.

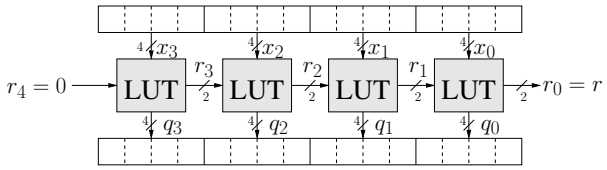


Fig. 3. LUT-based division by 3 of a 16-bit number written in radix 2^4 ($\alpha = 4$, $\gamma = 2$)

2.3 Memory structures in current FPGAs

Current FPGAs offer two main memory structures. The first is the 4- to 6- input LUT used in the logic fabric. In the following we note LUTk a k-bit input, 1-bit output LUT. In each FPGA family, there are restrictions on LUT utilization. Let us review recent FPGAs with the motivation to build k-input, k-output LUTs.

The Altera Stratix IV arithmetic and logic module (ALM) can be used as two arbitrary LUT4, but may also implement two LUT5 or two LUT6 under the condition that they share some of their inputs. This is the case for our architectures: a 6-input, 6-output LUT may be built as 3 ALMs.

In Xilinx Virtex-5 and Virtex-6, the logic slice includes 4 registers and 4 LUT6, each of which is fractionable as two LUT5 with independent outputs. The soft spot here is therefore to build 5-input tables, unless we need to register all the outputs, in which case 6-input tables should be preferred.

We may use, for instance, 6-input LUTs to implement division by 3 ($\gamma = 2$) in radix 16 ($\alpha = 4$). Implementing the core loop costs 6 LUTs (for a 6 bits in, 6 bits out table). The cost for a fully combinatorial (or unrolled) divider by 3 on n bits is $\lceil n/4 \rceil \times 6$ LUT6s, for instance 36 LUT6s for 24 bits (single precision), or 78 LUTs for 53 bits (double precision). The best shift-and-add algorithm to date needs respectively 118 and 317 full-adders (FA), each FA consuming one LUT both in Xilinx and in Altera devices. The approach proposed here is four times as efficient on division by 3. The larger d, the more inefficient this approach becomes, as we need more bits to represent the r_i.

For larger constants, a second option is the embedded memory block, from 9Kbits to 144 Kbits depending on the architecture. We will use them as $2^9 \times 9$

(9Kbits), $2^{10} \times 10$ (18Kbits or 36KBits) or $2^{13} \times 13$ (144 KBits). For division by 3, we may now use $\alpha = 7$ to $\alpha = 11$, but these larger memories also push the relevance of this technique to larger constants.

These memories are not combinatorial, their inputs must be registered: they are best suited to either sequential, or unrolled but pipelined implementation. In the latter case, we may exploit the fact that all these embedded memories are dual-ported: two iterations may be unrolled in one single memory block as depicted on Figure 4. Again for division by 3, exploiting the M9K blocks of a Stratix IV (using $\alpha = 7$), a fully pipelined single-precision divider by 3 could be impemented in 2 M9K only (2*2*7=28 bits) and run in 4 cycles at the maximal practical speed supported by these devices. We have no experimental data to support these claims as we implemented only the logic-based dividers so far. Indeed, results in Section 4 suggest that architectures based on embedded RAMs would not be very competitive.

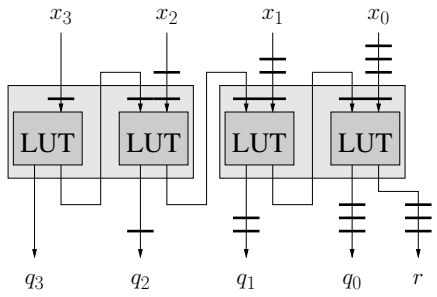


Fig. 4. A pipelined divider using two dual-ported embedded RAMs

3 Division of a floating-point number by a small integer constant

A floating-point input X is given by its mantissa m and exponent e:

$$x = 2^e m \quad \text{with } m \in [1, 2).$$

Similarly, the floating-point representation of our integer divisor d is:

$$d = 2^s d' \quad \text{with } d' \in [1, 2)$$

with $s = \gamma - 1$ if d is not a power of two.

As the mantissa has a fixed number of bits, its normalization and rounding have to be performed for almost each floating-point operation [8].

3.1 Normalization

Let us write the division

$$\frac{x}{d} = \frac{m.2^e}{d} = \frac{2^s m}{d} 2^{e-s}.$$

As $\frac{2^s m}{d} = \frac{m}{d'} \in [0.5, 2)$, this is almost the normalized mantissa of the floating-point representation of the result:

- if $m \geq d'$, then $\frac{m}{d'} \in [1, 2)$, the mantissa is correctly normalized and the floating-point number to be returned is

$$y = \circ\left(\frac{2^s m}{d}\right) 2^{e-s}$$

  where $\circ(z)$ denotes the IEEE-standard rounding to nearest even of a real z.
- if $m < d'$, then $\frac{m}{d'} \in [0.5, 1)$, the mantissa has to be shifted left by one. Thus, the floating-point number to be returned is

$$y = \circ\left(\frac{2^{s+1} m}{d}\right) 2^{e-s-1} \quad .$$

It can be observed that the comparison between m and d' is extremely cheap for small integers because d' has only γ non-zero bits. Thus, the comparison is reduced to the comparison of these γ bits to the leading γ bits of m. As both m and d' have a leading one, we need a comparator on $\gamma - 1$ bits. In terms of latency, this is a very small delay using fast-carry propagation.

3.2 Rounding

Let us now address the issue of correctly rounding the mantissa fraction. If we ignore the remainder, the obtained result is the rounding towards zero of the floating-point division.

To obtain correct rounding to the nearest, a first idea is to consider the final remainder. If it is larger than $d/2$, we should round up, *i.e.* increment the mantissa. The comparison to $d/2$ would cost nothing (actually the last table would hold the result of this comparison instead of the remainder value), but this would mean an addition of the full mantissa size, which would consume some logic and have a latency comparable to the division itself, due to carry propagation.

A better idea is to use the identity $\circ(z) = \lfloor z + \frac{1}{2} \rfloor$, which in our case becomes

$$\circ\left(\frac{2^{s+\epsilon} m}{d}\right) = \left\lfloor \frac{2^{s+\epsilon} m}{d} + \frac{1}{2} \right\rfloor = \left\lfloor \frac{2^{s+\epsilon} m + d/2}{d} \right\rfloor$$

with ϵ being 0 if $m \geq d'$, and 1 otherwise. In the floating-point context we may assume that d is odd, since powers of two are managed as exponents. Let us write $d = 2h + 1$. We obtain

$$\circ\left(\frac{2^{s+\epsilon} m}{d}\right) = \left\lfloor \frac{2^{s+\epsilon} m + h}{d} + \frac{1}{2d} \right\rfloor = \left\lfloor \frac{2^{s+\epsilon} m + h}{d} \right\rfloor$$

so instead of adding a round bit to the result, we may add h to the dividend before its input into the integer divisor. It seems we haven't won much, but this pre-addition is actually for free: the addend $h = \frac{d-1}{2}$ is an s-bit number, and we have to add it to the mantissa of x that is shifted left by $s + \epsilon$ bits, so it is a mere concatenation. Thus, we spare on the adder area and the carry propagation latency.

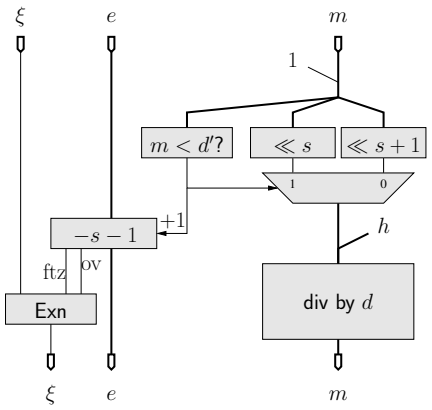


Fig. 5. Floating-point division by a small constant.

To sum up, the management of a floating-point input adds to the area and latency of the mantissa divider those of one (small) exponent adder, and of one (large) mantissa multiplexer, as illustrated by Figure 5. On this figure, ξ is a 2-bit exception vector used to represent 0, $\pm\infty$ and NaN (Not a Number).

The implementation in FloPoCo manages divisions by small integer constants and all their powers of two. The only additional issues are in the overflow/underflow logic (the Exn box on Figure 5), but they are too straightforward to be detailed here.

4 Results and comparison

All the results in this section are obtained for architectures generated by FloPoCo 2.3.0, using ISE 12.1 for an FPGA with 6-input LUTs (Virtex-5). These are synthesis results (before place and route).

Table 1 provides some results for Euclidean division (integer division with remainder). We only report the architecture obtained with the optimal value of α.

constant	$n = 32$ bits LUT6	(predicted)	latency	$n = 64$ bits LUT6	(predicted)	latency
$d = 3$ ($\alpha = 4, \gamma = 2$)	47	(6*8=48)	7.14ns	95	(6*16=96)	14.8ns
$d = 5$ ($\alpha = 3, \gamma = 3$)	60	(6*11=66)	6.79ns	125	(6*22=132)	13.8ns
$d = 7$ ($\alpha = 3, \gamma = 3$)	60	(6*11=66)	7.30ns	125	(6*22=132)	15.0ns

Table 1. Synthesis results for combinatorial Euclidean division on Virtex-5

4.1 Integer division

One would believe that for such simple architectures, we can predict the synthesis results, at least with respect to LUT count. However, there are still some surprises, which we are currently investigating.

The first surprise is that the synthesis tools perform further optimization out of our designs: the LUT numbers are not always those predicted (they are always better). For instance, for the 64-bit divider by 3, we predict simply 96 LUT6, but the tool reports 15 LUT3, 18 LUT4, 16 LUT5, and only 45 LUT6, then merges that into 95 LUTs. One of the reason could be that some remainder values never occur, which means that there are "don't care" in the logic tables that enable further optimizations. This would explain that the results are better for division by 5 than for division by 7 although they have the same α and β: there are more "don't care" in the table for 5. Such improvements should be studied systematically.

Also, we have mentioned earlier that the soft spot on Virtex-5 should be to use 5-input LUTs, but the synthesis tools seem to think otherwise: architectures designed for 5-input LUTs actually consume more than those designed for 6-input LUTs. This could come frome a coding style issue, or a misunderstanding of the intricate details of the Virtex-5 logic block.

Table 2 provides some synthesis results for pipelined dividers by 3. Each line is a different frequency/area tradeoff (incidentally, thanks to FloPoCo's pipelining framework [1], this flexible pipeline took less than ten minutes to implement out of the combinatorial design). Here we have to investigate why the LUT number is larger than the predicted size.

$n = 32$ bits		$n = 64$ bits	
FF + LUT6	performance	FF + LUT6	performance
33 Reg + 47 LUT	1 cycle @ 230 MHz	122 Reg + 112 LUT	2 cycles @ 217 MHz
58 Reg + 62 LUT	2 cycles @ 410 MHz	168 Reg + 198 LUT	5 cycles @ 410 MHz
68 Reg + 72 LUT	3 cycles @ 527 MHz	172 Reg + 188 LUT	7 cycles @ 527 MHz

Table 2. Synthesis results for pipelined Euclidean division by 3 on Virtex-5

4.2 Floating-point division

Table 3 shows results for floating-point division by 3. The behaviour of these operators, including the fact that they return correctly rounded results, has been verified by simulation against millions of test vectors generated by an independent floating-point division by 3 [1].

single precision		double precision	
FF + LUT6	performance	FF + LUT6	performance
35 Reg + 69 LUT	1 cycle @ 217 MHz	122 Reg + 166 LUT	2 cycles @ 217 MHz
105 Reg + 83 LUT	3 cycles @ 411 MHz	245 Reg + 250 LUT	6 cycles @ 410 MHz

Table 3. Synthesis results for pipelined floating-point division by 3 on Virtex-5

4.3 Comparison with previous work

A review of several algorithms for division by a constant is available in [6]. Many of these algorithms require the division to be exact (null remainder) and return wrong results otherwise. We will not consider them. Among the remaining techniques, the most relevant is method 6 in [6].

In this method, the divisor has the form $2^t \pm 1$, which corresponds to most of the small divisor we are targeting. The quotient and the remainder are obtained trough $\lceil \frac{n}{\gamma} \rceil - 1$ additions and substractions involving n-digit numbers.

Table 4 summarizes the comparison of the size of our implementation and an estimation of the area of an FPGA implementation of this previous technique. It can be observed that in this implementation, the larger d, the fewer required additions, therefore the smaller the implementation. This means that this method is increasingly relevant for larger constants $2^t \pm 1$, and this is a method to investigate in the future. Our proposition remains very significally smaller for small divisors.

	$n = 16$ bits		$n = 32$ bits		$n = 64$ bits	
Constant	Our	[6]	Our	[6]	Our	[6]
3	23	80	47	320	95	1344
5	29	48	60	192	125	768
7	29	32	60	128	125	640

Table 4. Comparison of the size in LUT between the implementation of our divider and [6] on Virtex 5

The presented floating-point division by a small constant also largely outperforms the best technique used so far, which are based on multiplication by

the constant $1/d$ using shift-and-add algorithm [4]. For instance, using this technique, a double-precision multiplication by 1/3, in the conditions of Table 3, consumes 282 reg + 470 LUT and runs in 5 cycles @ 307 MHz.

5 Conclusion

This article adds division by a small integer constant such as 3 or 10 to the bestiary of arithmetic operators that C-to-hardware compilers can use when they target FPGAs. This operation can be implemented very efficiently, be it for integer inputs, or for floating-point inputs. It is now part of the open-source FloPoCo generator.

Some synthesis results suggest that a careful study of the tables could lead to further optimizations. In addition, we should try to reformulate our tables so that the propagation of the r_i uses the fast-carry lines available on all modern FPGAs: this would reduce the latency dramatically.

Another issue worth of interest is the case of larger constants that are product of smaller constants, for which a cascaded implementation could be studied.

Due to routing pressure, the number of inputs to the FPGA LUTs keeps increasing as technology progresses. This should make this technique increasingly relevant in the future.

References

1. F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, Aug. 2011.
2. M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2003.
3. E. Artzy, J. A. Hinds, and H. J. Saal, "A fast division technique for constant divisors," *Communications of the ACM*, vol. 19, pp. 98–101, Feb. 1976.
4. F. de Dinechin, "Multiplication by rational constants," *IEEE Transactions on Circuits and Systems, II*, 2011, to appear.
5. S.-Y. R. Li, "Fast constant division routines," *IEEE Transactions on Computers*, vol. C-34, no. 9, pp. 866–869, Sep. 1985.
6. P. Srinivasan and F. Petry, "Constant-division algorithms," *IEE Proc. Computers and Digital Techniques*, vol. 141, no. 6, pp. 334–340, Nov. 1994.
7. R. W. Doran, "Special cases of division," *Journal of Universal Computer Science*, vol. 1, no. 3, pp. 67–82, 1995.
8. J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhauser Boston, 2009.

A Proof of correctness of Algorithm 1

The proof proceeds in two steps. First, we establish that $x = d\sum_{i=0}^{k} q_i.2^{-\alpha i} + r_0$ in lemma 1 below. This shows tha we compute some kind of Euclidean division, but it is not enough: we also need to show that the q_i form a binary representation of the result. For this it is enough to show that they are radix-2^α digits, which is established thanks to lemma 2 below.

Lemma 1.

$$x = d\sum_{i=0}^{k} q_i.2^{-\alpha i} + r_0$$

Proof. To show this lemma, we use the definition of the Euclidean division of y_i by d: $y_i = dq_i + r_i$.
$x = \sum_{i=0}^{k-1} x_i.2^{-\alpha i}$
$= \sum_{i=0}^{k-1}(x_i + 2^{\alpha} r_{i+1}).2^{-\alpha i} - \sum_{i=0}^{k-1}(2^{\alpha} r_{i+1}).2^{-\alpha i}$ and $r_k = 0$.
$= \sum_{i=0}^{k-1}(dq_i + r_i).2^{-\alpha i} - \sum_{i=1}^{k} r_i.2^{-\alpha i}$
$= d\sum_{i=0}^{k-1} q_i.2^{-\alpha i} + r_0 - r_k.2^{-\alpha k}$

Lemma 2. $\forall i \quad 0 \leq y_i \leq 2^{\alpha} d - 1$

Proof. The digit x_i verifies by definition $0 \leq x_i \leq 2^{\alpha} - 1$; r_{i+1} is either 0 (initialization) or the remainder of a division by d, therefore $0 \leq r_i \leq d - 1$. Therefore $y_i = x_i + 2^{\alpha} r_{i+1}$ verifies $0 \leq y_i \leq 2^{\alpha} - 1 + 2^{\alpha}(d-1)$, or $0 \leq y_i \leq 2^{\alpha} d - 1$.

We deduce from the previous lemma and the definition of q_i as quotient of y_i by d that

$$\forall i \quad 0 \leq q_i \leq 2^{\alpha} - 1$$

which shows that the q_i are indeed radix-2^{α} digits. Thanks to Lemma 1, they are the digits of the quotient.