Experimental Software and Hardware Evaluation of Ad-Hoc Constant Division Routines

Frédéric Pétrot
Univ. Grenoble Alpes, CNRS, Grenoble INP — UGA, TIMA, France
frederic.petrot@univ-grenoble-alpes.fr

***Abstract*—Dividing by a constant is an operation that is often needed in algorithms, be they implemented in software or in hardware. Given the fact that general purpose processors have had fast hardware multipliers for decades, the solution for software is to multiply by a compile time computed reciprocal and perform some adjustment. However, for hardware implementations, in particular with relatively small and exotic bit sizes, shift-and-add solutions might be worth looking at. In this paper, we report our study on implementing constant unsigned division based on Li's work. We found that a few of his algorithms are wrong and propose corrections that need a bit more computations. For software, we show that the approach can be useful only for low-end microcontrollers. For hardware, our FPGA and ASIC synthesis outline that it has good scalability, although being not very efficient for small dividends. As delay and area are very dependent on the value of the divisors, this approach appears as yet another possibility to choose from when looking on how to divide by a given constant.**

I. Introduction

We got interested in constant Euclidean division algorithms for hardware while looking on how to perform the *average pooling* operation that many convolutional neural networks need. In these networks, kernels are odd and square, thus 3×3, 5×5 and 7×7 kernels are vastly used (although larger sizes might also be used [1]). Average pooling is a new name for mean: it performs the sum of all values in the kernel and divides it by its size, that is a constant defined by the network architecture. Therefore, efficiently computing the quotient of an unsigned division by 9, 25 and 49 was our original goal.

Back in 1985, Li published a cookbook [2] of mostly non-trivial routines using shift-and-add to perform unsigned division for odd constants between 3 and 55. These routines have been found experimentally, and some of them are quite unexpected, but indeed work. Although useless for mainstream processors, as we will see, they might have a place for microcontrollers. They are also quite appealing for hardware in which shifts are free, and multipliers expensive. Our interest lies mainly in this latter use.

To the best of our knowledge, no systematic evaluation of Li's routines has been carried out, or at least reported in the literature. To start with, the paper introduces Section II Li's routines, focusing on our corrections of 5 of them and on the tiny modifications to do for their hardware versions. Then we detail our experiments. First, Section III, we compare the run times of Li's approach (SA, for shift-and-add) and Granlund and Montgomery (MH, for multiply high) [3] algorithm on a general-purpose processor. Second, Section IV, we synthesize circuits for odd constants between 3 and 55, for bit sizes between 4 and 32. We also compare to previous works on some constant values for dividend bit sizes up to 64. Finally, we summarize our findings and draw conclusions Section V.

II. Li's routines

In his paper, Li first presents a general approach to constant division based on the Fermat-Euler theorem, which states that if c and p are coprime, then $c^{\varphi(p)}$ is congruent to 1 modulo p, where φ is Euler's totient function. Assuming $c = 2$, $p > 1$ and odd, the theorem warranties the existence of a number $n < p$ such that p divides $2^n - 1$. He searches for the smallest such n and writes down $\frac{2^n-1}{p}$ in its binary form, $b_1b_2b_3\ldots b_{n-1}$. From that binary representation he derives the following equation:

$$\frac{1}{p} = 0.(0b_1b_2b_3\ldots b_{n-1})\prod_{i=0}^{\infty}(1+\frac{1}{2^{n\times2^i}})$$

Let us take the division by 23 to illustrate the principle. The smallest integer n such that 23 divides $2^n - 1$ is 11, and $2047/23 = 89$ or 1011001_2. The binary form has to be written on 11 bits to have the proper magnitude, leading to 00001011001_2. (Note that this is a nice theoretical way to find the base bit pattern in $\frac{1}{23}$ and its period). We now can write:

$$\frac{x}{23} = \left(\frac{x}{2^5}+\frac{x}{2^7}+\frac{x}{2^8}+\frac{x}{2^{11}}\right)\left(1+\frac{1}{2^{11}}\right)\left(1+\frac{1}{2^{22}}\right)\left(1+\frac{1}{2^{44}}\right)\cdots$$

For 32-bits values, this requires 6 shifts and 5 adds, while it would require 12 of them using $\frac{1}{23}$ raw binary decomposition. Although mathematically sound, this ignores truncation errors, and is not optimal. Li soon discovered that solutions with a lower count of operations can be found empirically on this basis "through trial and error", by in particular using subtraction and reuse of intermediate values. As opposed to more general solutions, his routines do not compute the remainder to make a final adjustment. These solutions are hard to derive, and Li gives a table that contains the best routines for odd divisors between 3 and 55 he could come out with. He reports brute force testing between 0 and 500×10^6 without errors.

Using his divide by 49 routine quickly led to an error that encouraged us to look more deeply into his cookbook, by basically retesting all routines. It happens that 5 routines out of 27 fail pretty quickly to properly divide 32-bit numbers. Three, for 7, 27 and 39, are typos, and two, 49 and 53 seem just wrong. Given the performances of the machines of that time, the "guessing" process was probably quite lengthy compared

2576-2265/25/$31.00 ©2025 IEEE
DOI 10.1109/ARITH64983.2025.00020

to what it is today, which partly explains the errors in the paper. Using an approach similar to Li's, we corrected the typos and devised two new routines for these cases. We give our corrections here in plain C.

Division by 7 misses one shift:

```
uint32_t divu7(uint32_t n)
{
   uint32_t x = n + 1;
   x = (x << 2) + (x >> 1);
   x = (x >> 6) + x;
   x = (x >> 12) + x; /* line missing in Li's paper */
   x = (x >> 24) + x;
   return x >> 5;
}
```

Division by 27 assigned the wrong variable on one line:

```
uint32_t divu27(uint32_t n)
{
   uint32_t x = n, y;
   y = (x << 1) + x + 15;
   x = (y >> 2) + (x << 2); /* y was assigned on this line */
   x = x - (x >> 9);
   x = (x >> 18) + x;
   return x >> 7;
}
```

Division by 39 added the wrong variable in one expression:

```
uint32_t divu39(uint32_t n)
{
   uint32_t x = n + 1, y;
   y = (x << 1) + x;
   y = (x >> 2) + y;
   x = (x >> 5) + y;    /* x was added on this line */
   x = (x >> 12) + x;
   x = (x >> 24) + x;
   return x >> 7;
}
```

We couldn't find simple syntactic changes in Li's division by 49 to correct it. We came out with the following routine, that uses the same number of additions but one more shift.

```
uint32_t divu49(uint32_t n)
{
   uint32_t x = n;
   x = (x << 2) - (x >> 5) + 2;
   x = x + (x >> 2) + (((x >> 4) + x) >> 4);
   x = (x >> 21) + x;
   return x >> 8;
}
```

Similarly to 49 for 53, we did not find what could be seen as typos, and determined the following routine. This routine uses two more additions and shifts than the erroneous original.

```
uint32_t divu53(uint32_t n)
{
   uint32_t x = n + 1, y;
   y = (x << 1) + (x >> 2);
   y = x + y + (y >> 5);
   y = y + ((x - (x >> 3)) >> 11) + ((x - (x >> 5)) >> 16)
       + (x >> 23) + (x >> 24);
   x = (x << 2) + (y >> 2);
   return x >> 8;
}
```

For software, these routines suffer from the fact that the dividend has first to be shifted left to avoid losing precious bits when later adding and shifting right. They must thus use the next larger type internally to avoid early errors. Nevertheless, if the upper bound of the dividend is known, which is often the case, then the routines can be used. For example, on 32-bit values, the bound at which the first of all routines starts to fail is $2200\ 0000_{16}$.

For hardware, adding 2 or 3 upper bits in an adder is not free, but far less than doubling its bit size. This makes the routines interesting in situations in which the bit size is constrained, as then the dividend maximal value is known beforehand.

III. Software usage

We measured the execution time of Li's routines (our modifications applied) and compared it to MH technique, as implemented in gcc v14.2.1. This latter method is fully general and exact but requires the computation of the upper n bits of the result of a $n \times n$ bits multiplication. All general-purpose processors implement this in hardware currently through the multiply high (unsigned in our case) instruction, with a multiplication throughput of 1 operation per cycle. The measure is performed by executing each constant division algorithm 10000 times, and then run again for the same number of iterations, but this time also measuring the elapsed time using the approach advocated by [4]. This ensures the cache is hot, which induces a low variance of the measurements. The computation is forced by qualifying the target quotient variable as `volatile` and by using the loop counter as dividend. This process is repeated 1000 times, and we compute a mean and a standard deviation over 100 instructions. The processor used for the experiments is an Intel i7-13800H. As visible Figure 1, the results are indisputable: MH outperforms significantly Li's approach in all but in 3 cases, the ones for which the adjustments after the multiply high are fairly complex (see Table I that reports the number of x86-64 instructions for the SA and MH methods), for which both solutions are in par. Interestingly enough, a division by a constant using MH takes in average just a bit more than one cycle on a modern core, half as much as the optimized SA approach.

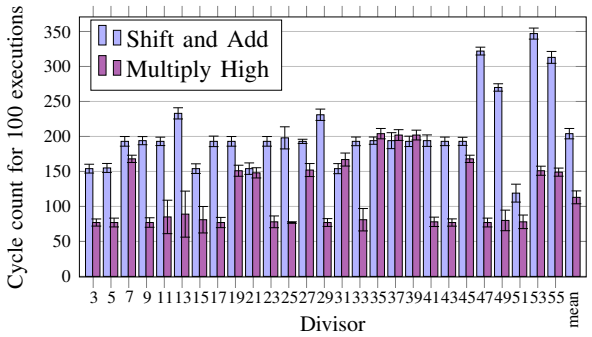

Fig. 1. Run times of Shift-and-Add and Multiply High, 32-bit dividend.

For small microcontrollers, e.g. based on the bare rv32i RISC-V instruction set or the ARM Cortex M0/M0+/M1 ones, the optimized SA approach can be useful. Although these latter processors might contain a 1 cycle 32×32 multiplication (The "slow" version of the instruction takes 32 cycles [5]), the 32 upper bits of the result needed for MH are not available, while computing them requires at least 16 basic instructions [6, Section 8.2]. Given our measures, constant division would then

TABLE I
NUMBER OF X86-64 INSTRUCTIONS FOR SA AND MH DIVISIONS (SA/MH)

divisor	3	5	7	9	11	13	15	17	19
nb insns	10/3	10/3	13/6	13/2	12/3	15/2	10/3	10/3	10/7
divisor	21	23	25	27	29	31	33	35	37
nb insns	10/7	13/3	12/2	9/6	13/3	10/6	12/2	13/7	11/7
divisor	39	41	43	45	47	49	51	53	55
nb insns	13/7	10/3	10/2	12/6	17/3	14/2	7/3	27/6	16/6

always be more efficiently done using SA on these architectures, even though, unlike the rest of the ARM cores, they do not support shifts within the `add` and `sub` instructions.

IV. HARDWARE USAGE

Doing constant division in hardware is quite common: filters in image processing accelerator are typical examples, but this spans many more areas. An authoritative source is de Dinechin and Kumm recent book [7, Chapter 13] that devotes a chapter to this topic. Several approaches are competing. One uses the paper and pencil algorithm using an appropriate power of two radix [8] to index look-up tables in series. This can be done partly in parallel, speeding-up the process at the cost of area [9]. A different parallel look-up table based approach followed by an adder tree is presented in [10]. These approaches are quite heavily based on small memory cuts, and compute both the quotient and reminder. Other approaches, such as Li's one, rely on bit patterns analysis and try to combine them through shifting in a way that minimizes the number of additions, e.g. [11, 12]. To the best of our knowledge there is no hardware synthesis result of Li's approach that has been published. Somehow, this is explainable, because the routines have been devised for relatively large dividends, are very ad-hoc, and fail before reaching the largest value that can be represented by the dividend type. However, in hardware we can extend the intermediate results with as many (in the order of 1 per assignment) bits as needed to support the full range of values of the type.

We translated Li's routines in Verilog (increasing the intermediate results), exhaustively tested them using Verilator, and ran FPGA synthesis using Vivado 23.2 for bit width ranging form 3 to 32, targeting AMD VC709 board. Figure 2 and 3 plots respectively the number of LUTs and the propagation delay as a function of the bit width for all divisors.

The Figures are a bit crowded, and not so easy to read due to the very ad-hoc nature of the computations: there is no reason why a smaller divisor would lead to a smaller (or bigger) circuit, as opposed to approaches that iteratively consume k bits to access a look-up table. Nevertheless, we can see on Figure 2 that the resources are increasing more or less linearly with the number of bits, and on Figure 3 that delays tend to plateau, with sudden increases when reaching a number of bits that depends on the algorithm.

To compare with the existing approaches, we made measurements at the same points than [9] and [10]. We present the results in Table II, in which the values relative to the existing approaches are borrowed from [10]. Note that quotient only

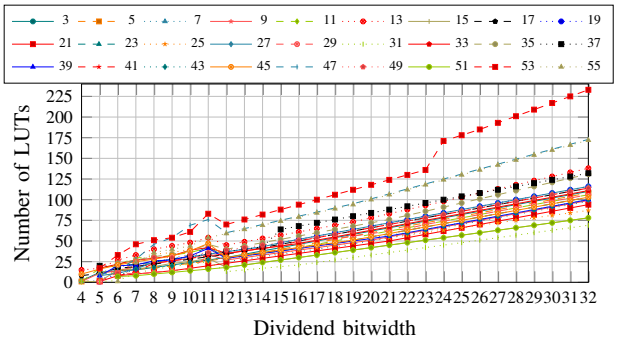

Fig. 2. Li: Number of LUTs on Virtex 7.

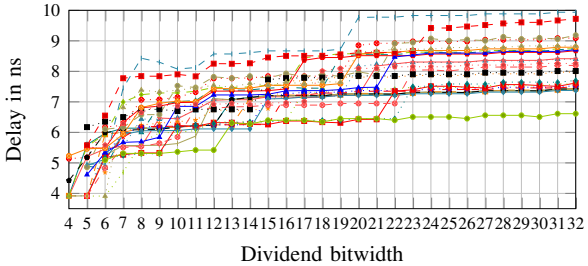

Fig. 3. Li: Circuit delay in ns on Virtex 7.

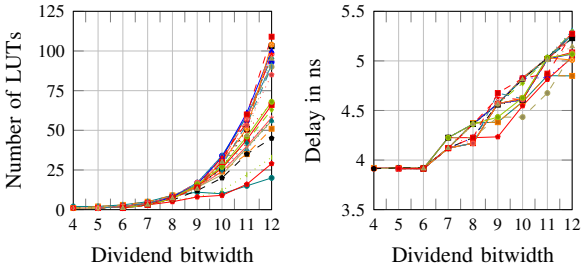

Fig. 4. Brute-force: Number of LUTs and delay on a Virtex 7.

computation is only very slightly simpler than computing the quotient and the remainder according to [9]. As Li's routines only produce the quotient, the comparison is a bit positively biased towards this method. In addition, the divisors 3, 5, 11 and 23 chosen in the previous works do not make much sense with the optimized SA approach, as the complexity in time and space of the solution depends on the bit pattern of the reciprocal and our capabilities to make use of it efficiently. Finally, although [9] synthesized for Kintex 7 while we do it for Virtex 7, the target LUTs are identical.

What can be seen on this Table is that Li's approach is not efficient for small dividend bit sizes compared to the other approaches, neither in area nor in speed. Nevertheless, the area and delay increase only slightly with the number of bits, which allows for a different trade-off than the other approaches.

We also generated constant tables synthesized as combinational circuits. The results of this brute-force approach are reported Fig. 4. Although we could synthesize tables with up to 24 address bits, the exponential growth in number of LUTs makes the approach unpractical above 11 bits.

For completeness, we synthesized ASIC versions of the

TABLE II
DELAY AND AREA AFTER FPGA SYNTHESIS
(BR: BLOCK RAM)

d	n	This work Delay ns	This work Area LUTs	SC [10] Delay ns	SC [10] Area LUTs	LinArch [9] Delay ns	LinArch [9] Area LUTs	BTCD [9] Delay ns	BTCD [9] Area LUTs + BR
3	16	7.2	46	4.1	40	3.6	17	3.7	37
	32	8.6	114	11.4	98	6.0	32	4.8	95
	64	9.0	277	27.5	379	13.5	63	6.2	225
5	16	7.2	44	4.0	52	4.4	21	3.8	44
	32	8.6	111	10.6	123	9.3	45	4.7	109
	64	9.0	274	27.3	386	20.1	93	6.7	270
11	16	7.5	42	3.9	53	8.0	39	3.8	79
	32	8.7	106	10.7	159	17.9	87	6.1	212
	64	10.4	265	26.9	436	39.0	183	8.8	526
23	16	7.5	41	3.9	52	7.4	69	5.6	197
	32	8.7	103	10.4	187	18.5	165	6.8	436 + 1BR
	64	10.4	256	26.1	493	36.6	357	6.5	959 + 2.5BR

circuits for the 4 above divisors, adding 47 and 51, that appear to be respectively the slowest and the fastest designs on FPGA. We target STMicroelectronics 28 nm FDSOI technology low power library, typical corner, 1 V, 40°C, with area optimization. Figures 5 and 6 plot respectively the area in μm^2 and the delay in ns of the circuits. Although we show the results for area optimization, delay optimization leads to very similar results.

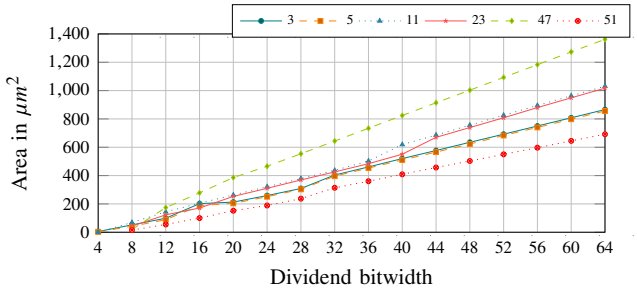

Fig. 5. Circuit area in μm^2 for STMicroelectronics 28 nm FDSOI technology.

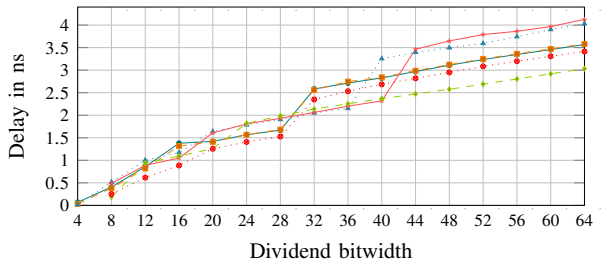

Fig. 6. Circuit delay in ns for STMicroelectronics 28 nm FDSOI technology.

We compared to [9] in which ASIC synthesis is done for the same technology. Li's routines are overall quite slower: they are quickly above a nanosecond, while the other works stay below, even though their slope is higher. The analysis shows long paths, due to the sequential nature of the adder tree the routines require. Area is on the other hand is around a factor of two below the other approaches.

As the previous works, these routines can be pipelined to reach a throughput of one division per cycle.

V. SUMMARY AND CONCLUSION

In this paper, we have thoroughly evaluated Li's fast constant division routines. First, we have checked their results and seen that, in 5 out of the 27 divisors of his cookbook (all odd numbers between 3 and 55 included), incorrect values were produced for 32-bit dividends. Three cases can be classified as typographic errors, while the last two were more challenging. We proposed corrections for all of them.

We then compared the approach of Li to the one of Granlund and Montgomery for software targets. We concluded that apart from small microcontrollers not providing the multiply high instruction, the latter approach is always better, by, in average, a factor of two. Next, we focused on hardware, which was our original intent. We synthesized the routines for both an FPGA and an ASIC target technology, and compared with alternative solutions. Overall, Li's routines offer good scalability but relatively large initial propagation delays, and their figures of merits are very dependent on the value of the divisor. They can be seen as one of a number of possibilities when it comes to choose a method to divide by a constant for a specific case.

The main drawback of these routines is their ad-hocness, and it turns out that on 64 bits, the 29, 35, 39, 49, 53 and 55 cases lead to errors much before expected due to rounding errors. A rather obvious perspective would be to try to formalize and generalize the approach, if only to guide a heuristic search.

Finally, the code that allows to reproduce this work is available at `https://github.com/fpetrot/divbysmallcst`.

ACKNOWLEDGMENTS

The author would like to acknowledge the financial support of the French ANR through MIAI@Grenoble Alpes ANR-19-P3IA-0003, and the EU and French DGE through the KDT JU EdgeAI project (N° 101097300).

REFERENCES

[1] Xiaohan Ding et al. "Scaling up your kernels to 31x31: Revisiting large kernel design in CNNs". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2022, pp. 11963–11975.

[2] S-YR Li. "Fast constant division routines". In: *IEEE Transactions on Computers* 100.9 (1985), pp. 866–869.

[3] Torbjörn Granlund and Peter L Montgomery. "Division by invariant integers using multiplication". In: *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. 1994, pp. 61–72.

[4] Gabriele Paoloni. *How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures*. Intel White Paper. 2010.

[5] ARM Ltd. *Cortex™-M0+ Technical Reference Manual*. 2012.

[6] Henry S. Warren Jr. *Hacker's delight*. Pearson Education, 2013.

[7] Florent de Dinechin and Martin Kumm. *Application-Specific Arithmetic: Computing Just Right for the Reconfigurable Computer and the Dark Silicon Era*. Springer, 2024.

[8] Florent De Dinechin and Laurent-Stéphane Didier. "Table-based division by small integer constants". In: *8th International Symposium on Reconfigurable Computing: Architectures, Tools and Applications*. Springer. Mar. 2012, pp. 53–63.

[9] H Fatih Ugurdag et al. "Hardware division by small integer constants". In: *IEEE Transactions on Computers* 66.12 (2017), pp. 2097–2110.

[10] Danila Gorodecky and Leonel Sousa. "Scalable architecture of constant division on FPGA". In: *IEEE 30th Symposium on Computer Arithmetic*. IEEE. 2023, pp. 16–23.

[11] Padmini Srinivasan and Frederick E. Petry. "Constant-division algorithms". In: *IEE Proceedings-Computers and Digital Techniques* 141.6 (1994), pp. 334–340.

[12] Florent De Dinechin. "Multiplication by rational constants". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 59.2 (2012), pp. 98–102.