On the Multiplierless Design of Correctly Rounded Multiple Constant Divisions

Levent Aksoy[†], Paulo Flores[†‡] and José Monteiro[†‡]
[†] INESC-ID, [‡] Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal

***Abstract*— The previously proposed algorithms designed for the constant divisions use the multiply-add architecture which is expensive in terms of area and power dissipation in hardware. This paper introduces the problem of finding the minimum number of adders/subtractors which realize the correctly rounded multiple constant divisions (MCD) under the shift-adds architecture. It proposes a depth-first search method which guarantees the minimum solution, exploring the whole search space. It also presents a local search method which can cope with the instances that the exact algorithm cannot handle due to the NP-completeness of the MCD problem. Both algorithms are equipped with techniques which maximize the sharing of common partial products among the constant multiplications. To the best of our knowledge, these are the first algorithms proposed for the multiplierless design of the correctly rounded MCD operation. Experimental results include the results of our algorithms, indicating that the local search method can find solutions close to the minimum using little computational resources and can obtain better solutions than a state-of-art algorithm. We also present the gate-level results of MCD designs under different architectures, showing that the MCD designs under the shift-adds architecture occupy less area and consume less power than those under the multiply-add architecture.**

I. Introduction

Constant division is not as common as constant multiplication in hardware and software applications generally because designers tend to avoid constant division due to its complexity in terms of area in hardware and cycle time in software. Hence, they realize the constant division x/d by multiplying the variable x by the approximation of the reciprocal of the divisor d, *i.e.*, $1/d$, or they modify d so that it can be realized using a shift and a few adders/subtractors. In both cases, a computational error is occurred. Nevertheless, constant division appears in forward and inverse quantization blocks of H.264/AVC [1], base conversions [2], and speech coding [3]. Also, compilers generate integer divisions to compute loop counts and subtract pointers [4].

Previously proposed constant division techniques can be grouped in two categories based on the application platform, *i.e.*, software and hardware. For constant divisions in software, the algorithm of [5] implements a constant division using a shift and a multiplier for the multiplication of the reciprocal of the divisor by the variable, where the reciprocals of constants are stored in a lookup table. Similarly, the algorithm of [6] approximates the reciprocal of the divisor and uses a multiplication and a couple of adjustment steps. The algorithm of [7] overcomes the approximation issues presented in [4], [5] under the multiply-add architecture, where $\lfloor x/d \rfloor$ is computed as $\lfloor (ax+b)/2^s \rfloor$. Observe that integer a approximates the scaled reciprocal $2^s/d$, integer b compensates for rounding errors, and integer s is an amount of right shift [7]. In [3], the proposed algorithms are also based on the multiply-accumulate instructions. For constant divisions in hardware, traditional techniques can be found in [8]. The shift-adds design of the multiplication of the variable x by a rational constant, which appears in the form of constant division, is introduced in [9]. Note that shifts by a constant can be implemented in hardware using only wires which represent no cost. In [10], the division by small integer constants using lookup tables is presented. In [11], the necessary and sufficient conditions for the realization of correctly rounded constant division under the multiply-add architecture for different rounding schemes are presented and finding a realization of the constant division, which requires the least hardware among possible alternatives, is formulated.

In this paper, we extend the work of [11] to multiple divisors and introduce the shift-adds design of correctly rounded multiple constant divisions (MCD) by replacing the constant multiplications by adders, subtractors, and shifts and maximizing the sharing of partial products. Although this paper focuses on the constant division under the round towards zero (RTZ) scheme, other rounding schemes such as, round to nearest, even and faithfully rounded schemes [11], can also be considered. Its main contribution is the introduction of optimization algorithms which find the smallest number of adders/subtractors that realize the correctly rounded MCD operation for the RTZ scheme.

II. Background

This section presents the background concepts and introduces the correctly rounded MCD problem.

A. Canonical Signed Digit Representation

The *binary* representation decomposes a number in a set of additions of powers of two. The signed digit system makes the use of positive and negative digits. The *canonical signed digit* (CSD) representation [12] has two main properties: (i) two nonzero digits are not adjacent; (ii) the number of nonzero digits is minimum.

Consider 23 defined in six bits. Its binary representation, 010111, includes 4 nonzero digits. It is represented as $10\bar{1}00\bar{1}$ in CSD using 3 nonzero digits, where $\bar{1}$ stands for -1.

B. Shift-Adds Design of Multiple Constant Multiplications

The multiple constant multiplications (MCM) operation realizes the multiplication of an input variable x by a set of

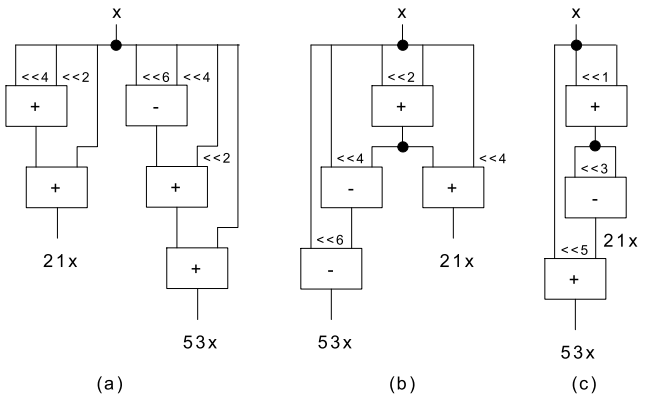


Fig. 1. Multiplierless realizations of $21x$ and $53x$: (a) DBR technique [8]; (b) exact CSE algorithm [14]; (c) exact GB algorithm [17].

constants. The digit-based recoding (DBR) method [8] is a straightforward way of realizing MCM without multipliers and has two steps: (i) define the constants under a number representation, *e.g.*, binary or CSD; (ii) for the nonzero digits in the representations of constants, shift the variable according to the digit positions and add/subtract the shifted variable with respect to the digit values. As a simple example, consider $21x$ and $53x$. The decompositions of constant multiplications under CSD are given as follows:

$$21x = (10101)_{CSD}x = x{\ll}4 + x{\ll}2 + x$$
$$53x = (10\bar{1}0101)_{CSD}x = x{\ll}6 - x{\ll}4 + x{\ll}2 + x$$

which lead to a design with 5 operations as shown in Fig. 1a.

Further reductions in the number of operations can be obtained by maximizing the sharing of partial products. Previously proposed algorithms can be grouped in two categories: (i) the *common subexpression elimination* (CSE) methods [12]–[14] define the constants under a particular number representation. They consider the possible subexpressions, which can be extracted from the nonzero digits in representations of constants, and choose the "best" subexpression, generally the most common, to be shared among the constant multiplications. Their main drawback is their dependency on a number representation; (ii) the *graph-based* (GB) techniques [15]–[17] are not restricted to any particular number representation and aim to find intermediate subexpressions which enable to realize the constant multiplications with a minimum number of operations. They consider a larger number of realizations of a constant and obtain better solutions than the CSE methods, but require more computational resources due to the larger search space. Note that finding the minimum number of adders/subtractors realizing the MCM operation is an NP-complete problem [18].

For our MCM example, the exact CSE algorithm [14] obtains a solution with 4 operations when constants are defined under CSD by finding the most common subexpression $5x = (101)_{CSD}x$ (Fig. 1b). The exact GB algorithm of [17] obtains a minimum solution with 3 operations by finding the intermediate subexpression $3x$ (Fig. 1c).

In algorithms proposed for the correctly rounded MCD problem, we prefer to use the GB methods of [16], [17] since they find better solutions than the CSE methods.

C. Correctly Rounded MCD under the RTZ Scheme

As described in [11], for the multiply-add realization of a single constant division under RTZ, $\lfloor x/d \rfloor = \lfloor (ax+b)/2^s \rfloor$ with $\forall x \in [0, 2^n - 1]$, where n denotes the bitwidth of the input variable x, a 3-tuple (a, b, s) must satisfy the necessary and sufficient conditions given as follows.

$$\begin{aligned} -(b{+}1)/\lfloor 2^n/d \rfloor < ad{-}2^s < a{-}b \quad &\text{if } ad{-}2^s < 0 \\ ad{-}2^s < (a{-}b)/\lfloor 2^n/d \rfloor \quad &\text{if } ad{-}2^s > 0 \end{aligned} \tag{1}$$

Thus, there may exist more than one 3-tuple for a constant divisor. Hence, we developed an exhaustive search algorithm to find all possible 3-tuples for each odd[1] constant d in between $[3, 2^n{-}1]$ under the given n value and stored them in a lookup table L^n. In this method, s ranges between $[0, 2n]$, otherwise there exist an infinite number of 3-tuples (a, b, s) which satisfy the conditions given in Eq. 1. Also, only odd values of the integer a are considered, since its even versions can be obtained using a left shift. During the search of 3-tuples for a single divisor, if there exist multiple 3-tuples with the same a value and different b values, the one, which has the minimum b value, is favored. This is because while a 3-tuple with $b{>}0$ requires the realization of ax and an adder, the one with b is zero only needs the realization of ax.

As an example, suppose d and n are 3 and 6, respectively. For the realization of $\lfloor x/3 \rfloor$ with 6-bit x values, there exist 43 possible 3-tuples. Five of them with the smallest s value are given as follows: (43, 0, 7); (85, 23, 8); (169, 107, 9); (171, 0, 9); (173, 0, 9). Observe that while the first 3-tuple requires the constant multiplication $43x$ to be realized, the second one needs $85x$ to be realized and an extra adder since b is greater than zero.

D. Problem Definition

The *correctly rounded MCD problem* can be defined as finding a minimum number of adders/subractors which implement the correctly rounded MCD operation. However, since each divisor requires alternative constant multiplications to be realized and may need an extra adder, it can also be defined as: given n, the set of divisors D, and the lookup table L^n which includes all possible 3-tuples of divisors, for each divisor in D, select a 3-tuple such that the multiplierless design of the correctly rounded MCD operation requires the minimum number of adders/subtractors.

III. Depth-First and Local Search Algorithms

This section presents the depth-first and local search methods proposed for the correctly rounded MCD problem. In their preprocessing phases, each integer divisor of D is converted to a positive number by multiplying it by -1 if it is negative and to an odd number by successively dividing it by 2 if it is even. The repeated constants and 1, which denotes the variable x, if they exist, are eliminated from D.

[1]The division of the variable x by an even constant e can be realized by its odd version o and a right shift with an amount of $r > 0$, where $e = o2^r$.

CASTOR(D, n, L^n)

1: $bs = \{\ \}$, $bc = \infty$
2: $depth = 0$, $ituple = [00\ldots0]$
3: $[T^n]$ = Configure(L^n, D)
4: **while** 1 **do**
5: [$depth$, $ituple$] = Branch($depth$, $ituple$, T^n, D)
6: **if** $depth \geq 2|D|/3$ **then**
7: lb = ComputeLowerBound($depth$, $ituple$, T^n, D)
8: **if** $lb < bc$ **then**
9: **if** $depth = |D|$ **then**
10: [$cost, sol$] = ComputeCost($ituple$, T^n)
11: **if** $cost < bc$ **then**
12: $bs = sol$, $bc = cost$
13: [$depth, ituple$] = Backtrack($depth$, $ituple$, T^n, D)
14: **if** $depth = 0$ **then**
15: **return** bs
16: **else**
17: [$depth, ituple$] = Backtrack($depth$, $ituple$, T^n, D)
18: **if** $depth = 0$ **then**
19: **return** bs

Fig. 2. Depth-first search algorithm.

A. Depth-First Search Algorithm

Pseudo-code of the depth-first search algorithm, called CASTOR, is given in Fig. 2, where bs and bc denote the best solution consisting of 3-tuples selected for the divisors of D and the best cost in terms of total number of adders/subtractors computed based on bs, respectively. The search tree constructed by CASTOR is shown in Fig. 3. The level in the search tree is denoted by $depth$, a node at each level is associated with a divisor of D, d_i with $1 \leq i \leq |D|$, and a branch denotes a possible assignment of a 3-tuple to a divisor. CASTOR traverses the search tree from top to bottom and left to right. Also, $ituple$ in the pseudo-code denotes an array with a size of $|D|$ and its each entry is the index of a 3-tuple assigned to a divisor of D in T^n which includes all possible 3-tuples for divisors of D.

Using the *Configure* function, CASTOR starts by storing all possible 3-tuples of each divisor of D in L^n to T^n and sorting these 3-tuples according to their $S(a) + sign(b)$ values in ascending order, where $S(a)$ denotes the number of nonzero digits of the integer a under CSD and $sign$ is the signum function. By doing so, CASTOR aims to find a solution close to the minimum in an earlier stage of the search that may enable to prune the search space in later stages. Then, in its infinite loop, the $depth$ value is increased by 1 and an assignment (a 3-tuple), which has not been considered, is made to the divisor at the current level using the *Branch* function. In this case, if the $depth$ value is greater than or equal to $2|D|/3$ [2], the lower bound on the number of adders/subtractors lb is computed based on the 3-tuples assigned so far using the *ComputeLowerBound* function. To do so, the technique of [19], which finds the lower bound on an MCM instance in terms of the number of adders/subtractors, is applied to all a values of the assigned 3-tuples and this value is incremented by 1 for each b value greater than zero in the assigned 3-tuples. If lb is greater than the best cost value found so far bc, the search is backtracked

[2]This value is determined empirically not to waste an effort for computing an estimate which usually does not yield a backtrack.

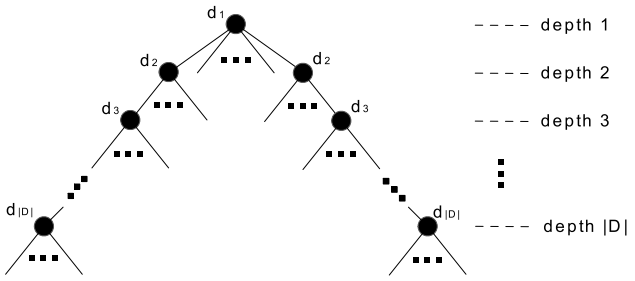


Fig. 3. Search tree constructed by CASTOR.

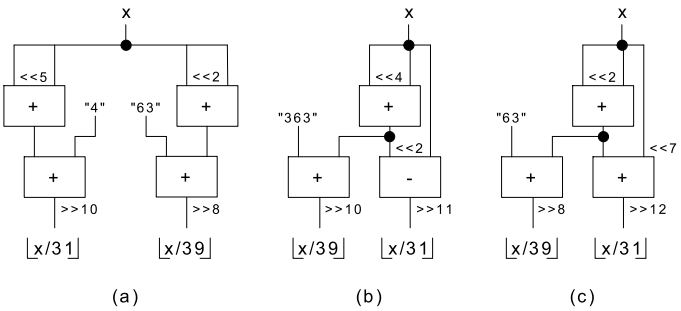


Fig. 4. Realizations of $\lfloor x/31 \rfloor$ and $\lfloor x/39 \rfloor$: (a) first solution of CASTOR; (b) final solution of CASTOR ; (c) final solution of POLLUX.

chronologically to a previous node until there exists at least one unassigned 3-tuple and an untried one is assigned to this node using the *Backtrack* function. Otherwise, if all divisors are assigned to a 3-tuple, the cost value of this set of 3-tuples $cost$ is found using the *ComputeCost* function. To do so, the exact GB algorithm of [17] is applied to an MCM instance consisting of all a values of 3-tuples for its multiplierless design, a minimum solution is found, and the number of adders/subtractors in this solution is incremented by 1 for each b value greater than zero in all 3-tuples. If this $cost$ value is smaller than bc, this set of 3-tuples and $cost$ are respectively assigned to bs and bc. CASTOR terminates whenever the $depth$ value is return to 0 indicating that all the search space is explored.

The performance of CASTOR depends heavily on the number of divisors, *i.e.*, $|D|$, and on the bitwidth of the input variable x, *i.e.*, n. As $|D|$ increases, the number of levels in the search tree increases and as n increases, the number of possible 3-tuples of a divisor increases, increasing the size of the search space to be explored. Also, as n increases, the values of a in 3-tuples increase, increasing the run-time of the exact MCM algorithm of [17].

As a simple example, suppose n is 6 and $D = \{31, 39\}$. The divisors 31 and 39 have 6 and 52 possible 3-tuples, respectively. The first solution of CASTOR requires 4 operations, where 3-tuples $(33, 4, 10)$ and $(5, 63, 8)$ are respectively assigned to the divisors 31 and 39, as illustrated in Fig. 4a. Observe that $33x$ and $5x$ are respectively implemented as $x \ll 5 + x$ and $x \ll 2 + x$ using 2 operations. However, its final solution includes 3-tuples $(67, 0, 11)$ and $(17, 363, 10)$ for the divisors 31 and 39, respectively, requiring a total of 3 adders/subtractors, as shown in Fig. 4b. Observe that $17x$ and $67x$ are realized using 2 operations as $x \ll 4 + x$ and $17x \ll 2 - x$, respectively.

B. Local Search Algorithm

Although CASTOR ensures the minimum number of operations, it can only be applied to MCD instances with a small number of divisors and a small n value. Thus, we propose a local search algorithm, called POLLUX, to handle a large number of divisors and n values using little computational resources. POLLUX does not consider all possible 3-tuples of divisors as CASTOR, but the promising ones which have the minimum $\lceil log_2 S(a) \rceil + sign(b)$ value. This comes from an observation that the 3-tuples with higher values were rarely found to be in the solution of CASTOR. Note that $\lceil log_2 S(a) \rceil$ is the lower bound on the number of adders/subtractors required for a single constant multiplication ax [19]. POLLUX also uses the heuristic method [16] rather than the exact method [17] used in CASTOR to find a multiplierless design of an MCM instance. Its pseudo-code is given in Fig. 5. In POLLUX, bs and bc have the same meanings as mentioned in CASTOR. Also, noi and noa denote the number of iterations and assignments, respectively and sol is a set with a size of $|D|$ and includes a 3-tuple for each divisor in D.

Using the *Determine3Tuples* function, POLLUX first determines the promising 3-tuples for each divisor in D as described earlier and sorts them as done in the *Configure* function of CASTOR. Second, it finds a solution to the MCD instance using the first 3-tuple of each divisor, computes the cost in terms of the number of adders/subtractors as described in CASTOR except using the MCM algorithm of [16], and assigns them to bs and bc, respectively in its *FindUpperBound* function. Then, it enters into an infinite loop. After sol is assigned to the best solution found so far bs, the 3-tuple of each divisor in sol is replaced by all possible 3-tuples of the divisor sequentially. The *ComputeLB* function finds the lower bound on sol as described in CASTOR and if this lower bound is smaller than the best cost value found so far bc, the *DetermineCost* function finds the cost of sol in terms of the number of adders/subtractors as described in CASTOR except using the MCM algorithm of [16]. If this cost value is smaller than bc, bs and bc are updated as sol and $cost$, respectively. The terminating conditions of its infinite loop are: (i) the number of iterations noi is equal to $|D|$; and (ii) the number of 3-tuple assignments noa is greater than $\prod_{i=1}^{|D|} |T^n(i)|$, which is the maximum number of assignments to be made in an exhaustive search method.

Returning to our example in Section III-A, POLLUX considers 5 and 7 3-tuples for the divisors 31 and 39, respectively. It finds a solution with a total of 4 operations using the *FindUpperBound* function when 3-tuples (33, 4, 10) and (5, 63, 8) are assigned to the divisors 31 and 39, respectively. Observe that this is the same as the first solution obtained by CASTOR illustrated in Fig. 4a. Its final solution includes 3 operations, where 3-tuples (133, 0, 12) and (5, 63, 8) are assigned to the divisors 31 and 39, respectively, as shown in Fig. 4c. These 3-tuples need the realizations of $5x$ and $133x$ which were found as $x \ll 2+x$ and $x \ll 7+5x$, respectively. On this example, POLLUX obtains a solution with the same number of operations as CASTOR.

POLLUX(D, n, L^n)

```
1:  bs = { }, bc = ∞
2:  [T^n] = Determine3Tuples(L^n, D)
3:  [bs, bc] = FindUpperBound(T^n, D)
4:  noi = 0, noa = 0
5:  while 1 do
6:      noi = noi + 1
7:      for i = 1 to |D| do
8:          for j = 1 to |T^n(i)| do
9:              noa = noa + 1
10:             sol = bs, sol(i) = T^n(i, j)
11:             lb = ComputeLB(sol)
12:             if lb < bc then
13:                 cost = DetermineCost(sol)
14:                 if cost < bc then
15:                     bs = sol, bc = cost
16:     if Terminating conditions are met then
17:         return bs
```

Fig. 5. Local search algorithm.

IV. Experimental Results

This section presents the results of the algorithm of [11], CASTOR, and POLLUX and of the gate-level designs of MCD operations realized under the multiply-add and shift-adds architectures. CASTOR and POLLUX were written in MATLAB and run on a PC with Intel Xeon at 2.33GHz and 10GB memory under a CPU time limit of 1200 seconds. The MCD operations were described in VHDL and synthesized using the Synopsys Design Compiler with the UMCLogic 180nm Generic II library. Their functionality was verified on 10,000 randomly generated input signals in simulation, from which the switching activity information, that was used by the synthesis tool to compute the power dissipation, was obtained.

For the comparison of CASTOR and POLLUX, we used randomly generated instances whose number of constants ranges in between 2 and 16 in steps of 2. Each group includes 30 instances and constants were generated in between $[1, 2^6-1]$. In this experiment, n was taken as 6. Table I presents the results of algorithms in terms of average number of adders/subtractors (*oper*) and run time in seconds (*cpu*).

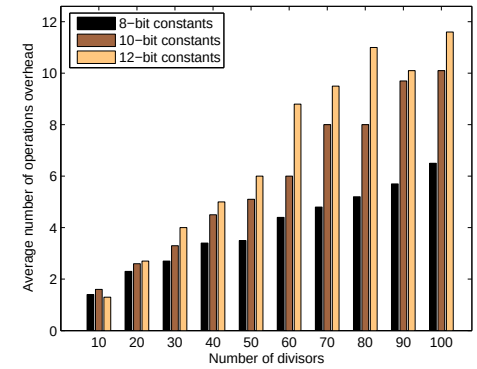
Observe from Table I that as the number of divisors increases, the difference on *oper* between POLLUX and CASTOR tends to increase, reaching up to 1 on MCD instances with 12 constants. However, in this case, the CPU time of CASTOR increases, reaching to the CPU time limit for all MCD instances with 16 constants which prevents CASTOR to guarantee the minimum solution. Thus, the difference on *oper* between POLLUX and CASTOR decreases as the number of divisors increases from 12 to 16. On the other hand, POLLUX uses little computational resources while finding the solutions of these MCD instances.

To compare the results of the state-of-art algorithm of [11] with POLLUX and to evaluate the performance of POLLUX, we generated 8-, 10-, and 12-bit constants randomly. The number of constants ranges in between 10 and 100 in steps of 10 and each group includes 30 instances. In this experiment, n was taken as 12. Fig. 6a presents the difference between the average number of adders/subtractors obtained by the algorithm of [11] and those found by POLLUX. To generate

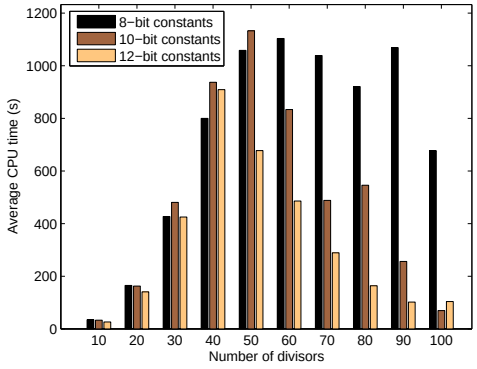
TABLE I

SUMMARY OF AVERAGE RESULTS OF ALGORITHMS ON 6-BIT RANDOMLY GENERATED CONSTANTS.

#divisors	2		4		6		8		10		12		14		16	
Algorithm	oper	cpu	oper	cpu	oper	cpu	oper	cpu	oper	cpu	oper	cpu	oper	cpu	oper	cpu
CASTOR	2.8	10.0	4.3	90.4	5.5	155.1	6.9	370.5	8.1	555.4	9.1	820.5	10.9	1166.4	11.9	1200.1
POLLUX	2.9	0.1	4.6	0.4	5.9	0.3	7.2	0.3	8.9	0.1	10.1	0.1	11.5	0.1	12.3	0.1



(a)



(b)

Fig. 6. (a) Difference between the average number of operations obtained by the algorithm of [11] and those found by POLLUX; (b) average CPU time of POLLUX.

more competitive results of the algorithm of [11] with respect to POLLUX, for each divisor in the MCD instance, among its possible 3-tuples, we selected a 3-tuple with a minimum $\lceil log_2 S(a) \rceil + sign(b)$ value and we used the MCM algorithm of [16], which is also used in POLLUX, to realize the constant multiplications in these 3-tuples under the shift-adds architecture. Furthermore, Fig. 6b presents the average CPU time of POLLUX. We note that the solutions of the algorithm of [11] were obtained in less than a second on average.

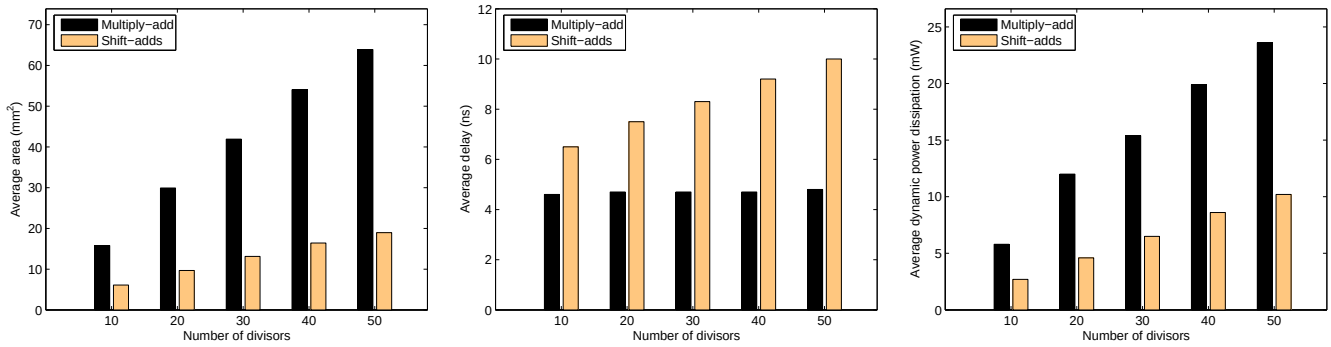
Observe from Fig. 6a that as the number of divisors increases, the difference between the average number of operations obtained by the algorithm of [11] and those found by POLLUX increases, reaching up to 11.6 on instances including 100 12-bit divisors. This is simply because while POLLUX considers many possible 3-tuples for a divisor, the algorithm of [11] considers only one 3-tuple for a divisor. Also, this difference tends to increase as the bitwidth of divisors increases. This is due to the fact that the number of adders/subtractors in the MCM design increases as the bitwidth of constants increases [17]. In turn, for all randomly generated instances under different bitwidth of constants, the CPU time of POLLUX tends to increase, as the number of divisors increases till it is 40. For 10- and 12-bit constants, its CPU time tends to decrease as the number of divisors increases from 50 and 40 to 100, respectively. This is mainly because larger divisors close to $2^n - 1$ include 3-tuples with small a values which help POLLUX to find a solution with the smallest number of operations in an earlier iteration.

For the comparison of MCD operations designed under the multiply-add and shift-adds architectures, we used the randomly generated instances including 8-bit integers, where the number of divisors ranges in between 10 and 50 in steps of 10 and each group includes 30 instances. In this experiment, n was taken as 12. For the multiply-add architecture, for each divisor in an MCD instance, a 3-tuple including the minimum $\lceil log_2 a \rceil$ value with the minimum b value is found and it is described as a constant multiplication and addition with a right shift in VHDL. For the shift-adds architecture, each MCD instance is described in VHDL based on the solution of POLLUX. Fig. 7 presents the average results of MCD designs in terms of area in mm^2, delay of the critical path in ns, and dynamic power dissipation in mW.

Observe from Fig. 7a that the shift-adds architecture based on the solution of POLLUX leads to MCD designs with significantly less area than those designed under the multiply-add architecture. On MCD instances including 50 divisors, the average area of MCD designs under the multiply-add architecture is 3.3 times larger than that of MCD designs under the shift-adds architecture. However, the MCD designs under the multiply-add architecture has less delay than those under the shift-adds architecture. As the number of divisors increases, the delay value under the multiply-add architecture increases slightly. This is because the maximum size of a multiplier, which has a significant impact on the critical path of the MCD design, is almost the same in MCD instances including different number of divisors. However, the delay increases considerably under the shift-adds architecture as the number of divisors increases. This is because the sharing of partial products in the shift-adds architecture increases in this case which increases the number of adder-steps, *i.e.*, the maximum number of operations in series [20]. On the other hand, the shift-adds architecture leads to MCD designs that consume significantly less power when compared to the multiply-add architecture, since its designs occupy less area. On MCD instances including 50 divisors, the average power dissipation of MCD designs under the multiply-add architecture is 2.3 times larger than that of MCD designs under the shift-adds architecture.

V. CONCLUSIONS

This paper introduced the correctly rounded MCD problem and proposed exact and approximate algorithms to find the minimum number of adders/subtractors which realize the



(a) (b) (c)

Fig. 7. Results of MCD designs under the multiply-add and shift-adds architectures: (a) area; (b) delay; (c) power.

MCD operation. It presented the results of algorithms on randomly generated instances and introduced the gate-level designs of MCD operations under the multiply-add and shift-adds architectures. It was shown that while the exact algorithm can only be applied to MCD instances including small number of divisors and small bitwidth of the input variable, the approximate algorithm can be applied to a large number of divisors, obtains better solutions than the state-of-art algorithm, and finds a solution using little computational resources. It was also observed that the shift-adds architecture yields MCD designs that occupy less area and consume less power than the multiply-add architecture, but having a larger delay.

VI. Acknowledgment

This work was supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under the project PEst-OE/EEI/LA0021/2013.

References

[1] T. Dias, N. Roma, and L. Sousa, "High Performance Unified Architecture for Forward and Inverse Quantization in H.264/AVC," in *Proc. of EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools*, 2012, pp. 632–639.

[2] R. W. Doran, "Special Cases of Division," *Journal of Universal Computer Science*, vol. 1, no. 3, pp. 67–82, 1995.

[3] J.-M. Muller, A. Tisserand, B. de Dinechin, and C. Monat, "Division by Constant for the ST100 DSP Microprocessor," in *Proc. of IEEE Symposium on Computer Arithmetic*, 2005, pp. 124–130.

[4] T. Granlund and P. Montgomery, "Division by Invariant Integers Using Multiplication," in *Proc. of the SIGPLAN94 Conference on Programming Language Design and Implementation*, 1994, pp. 61–72.

[5] R. Alverson, "Integer Division Using Reciprocals," in *Proc. of IEEE Symposium on Computer Arithmetic*, 1991, pp. 186–190.

[6] N. Möller and T. Granlund, "Improved Division by Invariant Integers," *IEEE Tran. on Computers*, vol. 60, no. 2, pp. 165–175, 2011.

[7] A. Robison, "N-Bit Unsigned Division via N-Bit Multiply-Add," in *Proc. of IEEE Symposium on Computer Arithmetic*, 2005, pp. 131–139.

[8] M. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2003.

[9] F. de Dinechin, "Multiplication by Rational Constants," *IEEE Tran. on Circuits and Systems II*, vol. 59, no. 2, pp. 98–102, 2012.

[10] F. de Dinechin and L.-S. Didier, "Table-Based Division by Small Integer Constants," in *Proc. of the International Symposium on Applied Reconfigurable Computing*, 2012, pp. 53–63.

[11] T. Drane, W.-C. Cheung, and G. Constantinides, "Correctly Rounded Constant Integer Division via Multiply-Add," in *Proc. of IEEE International Symposium on Circuits and Systems*, 2012, pp. 1243–1246.

[12] R. Hartley, "Subexpression Sharing in Filters Using Canonic Signed Digit Multipliers," *IEEE Tran. on Circuits and Systems II*, vol. 43, no. 10, pp. 677–688, 1996.

[13] I.-C. Park and H.-J. Kang, "Digital Filter Synthesis Based on Minimal Signed Digit Representation," in *Proc. of Design Automation Conference*, 2001, pp. 468–473.

[14] L. Aksoy, E. Costa, P. Flores, and J. Monteiro, "Exact and Approximate Algorithms for the Optimization of Area and Delay in Multiple Constant Multiplications," *IEEE Tran. on Computer-Aided Design of Integrated Circuits*, vol. 27, no. 6, pp. 1013–1026, 2008.

[15] A. Dempster and M. Macleod, "Use of Minimum-Adder Multiplier Blocks in FIR Digital Filters," *IEEE Tran. on Circuits and Systems II*, vol. 42, no. 9, pp. 569–577, 1995.

[16] Y. Voronenko and M. Püschel, "Multiplierless Multiple Constant Multiplication," *ACM Tran. on Algorithms*, vol. 3, no. 2, 2007.

[17] L. Aksoy, E. Gunes, and P. Flores, "Search Algorithms for the Multiple Constant Multiplications Problem: Exact and Approximate," *Elsevier Journal on Microprocessors and Microsystems*, vol. 34, no. 5, pp. 151–162, 2010.

[18] P. Cappello and K. Steiglitz, "Some Complexity Issues in Digital Signal Processing," *IEEE Tran. on Acoustics, Speech, and Signal Processing*, vol. 32, no. 5, pp. 1037–1041, 1984.

[19] O. Gustafsson, "Lower Bounds for Constant Multiplication Problems," *IEEE Tran. on Circuits and Systems II*, vol. 54, no. 11, pp. 974–978, 2007.

[20] H.-J. Kang and I.-C. Park, "FIR Filter Synthesis Algorithms for Minimizing the Delay and the Number of Adders," *IEEE Tran. on Circuits and Systems II*, vol. 48, no. 8, pp. 770–777, 2001.