Bit-Level Optimized Constant Multiplication using Boolean Satisfiability

Nicolai Fiege, Martin Kumm, *Member, IEEE*, Peter Zipf, *Member, IEEE*

***Abstract*—Multiplierless constant multiplication using bit-shifts, additions and subtractions has been an active research topic in the last decades. The multiplication with multiple constants, known as the multiple constant multiplication (MCM) problem, is of special interest because of its practical relevance, notably for digital filter implementation. In this work we propose to use the speed of modern Boolean satisfiability (SAT) solvers to find fast and optimal solutions. The solutions are optimal either with respect to the adder count or the bit level cost. In contrast to previous approaches, we also consider negative fundamentals that are sometimes cheaper to realize than their positive counterparts leading to more compact hardware implementations. Our experiments show that our approach is able to find optimal single constant multiplication (SCM) and MCM circuits for practically relevant test instances in reasonable time. We also prove the necessity for the post-add right shift operation for SCM. Using our SAT formulation to enumerate all possible implementations for some of our test instances we show the importance of considering bit-level costs and negative fundamentals when solving MCM problems.**

I. Introduction

Multiplication by a constant (i.e., the scaling operation) is a common operation in nearly all numeric algorithms and can be efficiently realized using only shift-and-add/subtract operations. However, finding a realization with a minimal number of additions (subtractions are counted as additions) was shown to be an NP-complete optimization problem [1], [2], even for the elementary case of a single constant, called single constant multiplication (SCM). The generalization to multiple constant multiplication (MCM) is of special practical relevance because of its application to digital filters and discrete transforms.

Finding a circuit with the least amount of adders for MCM has long been an active research topic because it promises to reduce resource requirements in the final hardware implementation [3]–[7]. Nevertheless, the choice of operations (i.e., bit shifts and addition vs. subtraction) heavily influences the used resources, even for instances with the same adder count. Therefore, a similar yet even harder optimization goal is to directly minimize bit-level cost metrics such as the number of full adders or the number of basic logic elements on a Field-Programmable Gate Array (FPGA) [8]–[12]. Previous work assumes that ripple carry adders (RCA) are used to implement additions [10]–[14]. Moreover, RCAs are the default adder topology for FPGA implementations, which is why our focus lies on RCA word size reduction within this work. As our approach targets bit-width reductions, results can be in principle transferred to other adder types, although concrete savings might slightly vary due to structural differences.

N. Fiege and P. Zipf are with the University of Kassel, Germany.
M. Kumm is with the Fulda University of Applied Sciences, Germany.
Manuscript received ???,???; revised August ???, ???.

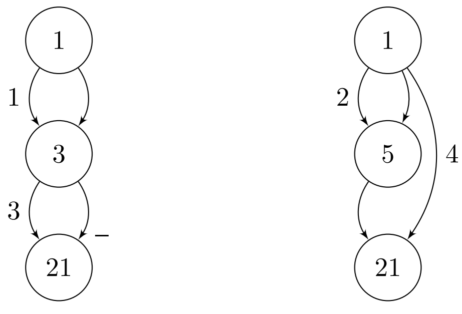
Multiplication by a constant is a well-known optimization problem in computer science. SCM being NP-complete [1], [2] means that optimally solving SCM or MCM instances takes exponential time in the worst case. Nevertheless, several optimal SCM and MCM algorithms have been published that are able to handle practically relevant problem sizes [4]–[7], [10]–[12], [15]–[18].

Apart from counting adders as a high-level optimization goal, the implementation complexity of each of the adders strongly depends on the word size, bit shifts and whether it is an addition or subtraction. For example, adding a number y to a shifted number x (i. e., $z = 2^s x + y$) the s lowest bits of z are identical to the s lowest bits of y and do not need any full adders. Considering these bit-level cost in the MCM problem was first proposed by Johansson et al. [8], [9] and later used by Aksoy et al. [10]. Their models consider the bit shifts of all the cases that can occur in an adder graph leading to an accurate number of full adders, half adders and even inverters [10]. A bit-level model can also be used to reduce the critical path delay as demonstrated by Lou et al. [13], [14]. Gustafsson et al. showed how to avoid sign extensions in MCM implementations [19]. All these previous works are heuristic optimizations. Two recent works addressed the optimal design of MCM considering bit-level cost at a full-adder level [11], [12] using ILP.

Optimizing circuits in a closed mathematical framework such as ILP has the advantage that solutions are proven to be optimal[1], and, depending on the framework, can easily be extended towards other optimization criteria or constraints. While ILP can suffer from numerical problems, especially for constant multiplication models with large coefficient word sizes [4] (cf. Section VIII for examples), SAT turned out to be a powerful framework to tackle various decision and optimization problems in electronic design automation that does not suffer from numerical issues since the problem is expressed as a Boolean formula. SAT solvers emerged to powerful tools in the recent years and have been successfully applied to, e.g., standard cell routing [20], layout synthesis for CMOS logic cells [21], logic synthesis [22], FPGA routing [23]–[25], verification [26], [27], and even SCM [7].

The only work that used SAT so far to tackle the constant multiplication problem using shift-and-add was proposed by Lagoon and Metodi [7]. They proposed to formulate the SCM

[1]under the assumptions of the problem formulation



(a) Non-optimal bit-level costs (b) Optimal bit-level costs

Fig. 1: Adder graphs to compute $y = 21 \cdot x$

problem in SystemVerilog and convert it using an automated procedure to a set of clauses that can afterwards be processed using a standard SAT solver [7]. While results are promising and outperform the state-of-the-art at that time, their approach suffers from several limitations:

- No post-add shift operation is allowed, which does not allow an optimal solution w.r.t. adder count in the general case.
- The clause generation is done by a proprietary tool, which does not give insight into the underlying SAT problem.
- MCM is not supported.
- Only optimality w.r.t. adder count is guaranteed.

Addressing the above mentioned shortcomings, our contributions are the following:

- We propose an algorithm based on Boolean Satisfiability (SAT) to solve the SCM (Section IV) and MCM (Section V) problems optimally regarding the adder count.
- We extend it to also support multiplication by negative numbers (Section VI) in preparation for bit-level cost optimizations.
- We extend it towards optimizing bit-level costs (Section VII).
- We experimentally validate that our algorithm can be applied to practically relevant problem sizes (Section VIII).

Our software is available as an open source project under https://doi.org/10.48662/daks-21.

II. Motivating example

Consider the constant multiplication by $C = 21$. Fig. 1 shows two possible solutions requiring the optimal number of adders, namely two. The circuit descriptions are given as a so-called *adder graph*. Each node in an adder graph represents an adder or subtractor, leading to a multiplication of the input by the node's constant and edge weights represent left shift operations. Hence, node "1" corresponds to the input of the circuit. For example, in Fig. 1a, node "3" corresponds to the computation of $3x$ by using $3x = (x \ll 1) + x$ while the node "21" computes $21x = (3x \ll 3) - 3x$. Fig. 1b shows an alternative graph using the same adder count. Table I gives a summary of our notation for constants and variables used in the following.

Both circuits shown in Fig. 1 need to implement two adders but they still differ in bit-level costs. The reason for this is

TABLE I: Describing the SCM/MCM problem: Constants (first part); node input/output decision variables (second part); node internal decision variables for minimum adder count (third part); and helper variables for bit optimization (fourth part)

Constant/variable	Explanation
$C^{(m)} \in \mathbb{N}$	target constants (MCM)
$M \in \mathbb{N}$	number of target constants ($M = 1$ for SCM)
$N \in \mathbb{N}$	number of adders
$N_{\mathrm{LB}} \in \mathbb{N}$	lower bound for the number of adders
$S = \max_m \lceil \log_2 C^{(m)} \rceil$	max. allowed shift
$W = S + 1$	internal word size within the SAT formulation
$W_{\mathrm{in}} \in \mathbb{N}$	input word size
$\sigma = \lceil \log_2 W \rceil$	shift word size
$\hat{B} \in \mathbb{N}$	upper limit for bit-level costs
$W_{\mathrm{B}} \in \mathbb{N}$	bit-level cost word size
$c^{(i)} \in \mathbb{N}$	output of node i
$t^{(i,m)} \in \{0, 1\}$	indicates whether $c^{(i)} = C^{(m)}$ (only for MCM)
$\alpha^{(i)} \in \mathbb{N}$	select input of left input MUX of node i
$\beta^{(i)} \in \mathbb{N}$	select input of right input MUX of node i
$\gamma^{(i)} \in \mathbb{N}$	pre-add shift input value of node i
$\delta^{(i)} \in \{0, 1\}$	negate input MUX select bit of node i
$\varepsilon^{(i)} \in \{0, 1\}$	$\overline{\text{add}}$/sub select bit of node i
$\zeta^{(i)} \in \mathbb{N}$	post-add shift input value of node i
$l^{(i)} \in \mathbb{N}$	left input MUX output of node i
$r^{(i)} \in \mathbb{N}$	right input MUX output of node i
$s^{(i)} \in \mathbb{N}$	pre-add shift output of node i
$x^{(i)} \in \mathbb{N}$	left negate select MUX output of node i
$u^{(i)} \in \mathbb{N}$	right negate select MUX output of node i
$y^{(i)} \in \mathbb{N}$	negate output of node i
$z^{(i)} \in \mathbb{N}$	adder output of node i
$a^{(i)} = \lvert z^{(i)} \rvert$	absolute value of the adder output
$v^{(i)} = \lceil \log_2(b^{(i)}) \rceil$	word size of node i's adder output
$m^{(i)} \in \{0, 1\}$	whether the bit for computing node i's MSB can be omitted
$g^{(i)} \in \mathbb{N}$	the number of bits omitted on the LSB side due to the shift operation

depicted in Table II. The addition to construct $3x$ needs to perform 6 bit operations for an input word size $W_{\mathrm{in}} = 6$ and assuming a signed input using a two's complement representation. We represent all integer variables (e.g., $x^{(i)}$ in Table II) as binary numbers and add the subscript w whenever we refer to bit w of those variables (e.g., $x_w^{(i)}$). Unless stated otherwise, the formulas/clauses provided from this point forward apply to all bits w of these integer variables. With *bit operation* we refer to the processing of single bits of the same weight either without a carry (half adder) or with carry (full adder) that may include inversion in case of a subtraction. The result has a word size of 8 bits but the computation for the LSB does not require any hardware because $2x$ always has a trailing zero independent of the actual input value x due to the shift operation. Furthermore, the computation for the MSB can also be omitted because the result always has the same sign as the input, as shown in Table IIa. When computing $21x = (3x \ll 3) - 3x$, we cannot leave out the bit-level computations for the lowest 3 bits, because it is necessary to perform the inversion and handle the carry input needed for subtraction. Therefore, 10 bits must be computed in hardware in this case, which results

TABLE II: Bit-level summations for $21x$ and a word size of 6 bits; triangles denote positions where a bit operation is needed

(a) $3x = (x \ll 1) + x$

		▽	▽	▽	▽	▽	▽	
$2x$	$x_5^{(1)}$	$x_5^{(1)}$	$x_4^{(1)}$	$x_3^{(1)}$	$x_2^{(1)}$	$x_1^{(1)}$	$x_0^{(1)}$	0
x	$x_5^{(1)}$	$x_5^{(1)}$	$x_5^{(1)}$	$x_4^{(1)}$	$x_3^{(1)}$	$x_2^{(1)}$	$x_1^{(1)}$	$x_0^{(1)}$
$3x$	$x_7^{(3)}$	$x_6^{(3)}$	$x_5^{(3)}$	$x_4^{(3)}$	$x_3^{(3)}$	$x_2^{(3)}$	$x_1^{(3)}$	$x_0^{(3)}$

(b) $21x = (3x \ll 3) - 3x$

		▽	▽	▽	▽	▽	▽	▽	▽	▽	▽
$24x$	$x_7^{(3)}$	$x_6^{(3)}$	$x_5^{(3)}$	$x_4^{(3)}$	$x_3^{(3)}$	$x_2^{(3)}$	$x_1^{(3)}$	$x_0^{(3)}$	0	0	0
$\overline{3x}$	$\overline{x}_7^{(3)}$	$\overline{x}_7^{(3)}$	$\overline{x}_7^{(3)}$	$\overline{x}_7^{(3)}$	$\overline{x}_6^{(3)}$	$\overline{x}_5^{(3)}$	$\overline{x}_4^{(3)}$	$\overline{x}_3^{(3)}$	$\overline{x}_2^{(3)}$	$\overline{x}_1^{(3)}$	$\overline{x}_0^{(3)}$
c_{in}											1
$21x$	$x_{10}^{(21)}$	$x_9^{(21)}$	$x_8^{(21)}$	$x_7^{(21)}$	$x_6^{(21)}$	$x_5^{(21)}$	$x_4^{(21)}$	$x_3^{(21)}$	$x_2^{(21)}$	$x_1^{(21)}$	$x_0^{(21)}$

(c) $5x = (x \ll 2) + x$

		▽	▽	▽	▽	▽	▽		
$4x$	$x_5^{(1)}$	$x_5^{(1)}$	$x_4^{(1)}$	$x_3^{(1)}$	$x_2^{(1)}$	$x_1^{(1)}$	$x_0^{(1)}$	0	0
x	$x_5^{(1)}$	$x_5^{(1)}$	$x_5^{(1)}$	$x_5^{(1)}$	$x_4^{(1)}$	$x_3^{(1)}$	$x_2^{(1)}$	$x_1^{(1)}$	$x_0^{(1)}$
$5x$	$x_8^{(5)}$	$x_7^{(5)}$	$x_6^{(5)}$	$x_5^{(5)}$	$x_4^{(5)}$	$x_3^{(5)}$	$x_2^{(5)}$	$x_1^{(5)}$	$x_0^{(5)}$

(d) $21x = (x \ll 4) + 5x$

		▽	▽	▽	▽	▽	▽				
$16x$	$x_5^{(1)}$	$x_5^{(1)}$	$x_4^{(1)}$	$x_3^{(1)}$	$x_2^{(1)}$	$x_1^{(1)}$	$x_0^{(1)}$	0	0	0	0
$5x$	$x_8^{(5)}$	$x_8^{(5)}$	$x_8^{(5)}$	$x_7^{(5)}$	$x_6^{(5)}$	$x_5^{(5)}$	$x_4^{(5)}$	$x_3^{(5)}$	$x_2^{(5)}$	$x_1^{(5)}$	$x_0^{(5)}$
$21x$	$x_{10}^{(21)}$	$x_9^{(21)}$	$x_8^{(21)}$	$x_7^{(21)}$	$x_6^{(21)}$	$x_5^{(21)}$	$x_4^{(21)}$	$x_3^{(21)}$	$x_2^{(21)}$	$x_1^{(21)}$	$x_0^{(21)}$

in 16 bit operations for the whole circuit.

The optimal implementation (cf. Fig. 1b) needs to perform 6 bit operations to realize $5x = (x \ll 2) + x$ (Table IIc) and also only 6 bits to compute $21x = (x \ll 4) + 5x$ (Table IId), resulting in only 12 bit operations in total. Therefore, we are able to cut 4 bit operations from the non-optimal circuit by choosing a different adder graph with the same adder count.

III. Proposed algorithm

We propose to use the search procedure outlined in Algorithm 1 to find an adder graph with optimal bit-level costs. The algorithm consist of two parts: (1) first, we compute a solution for the optimal adder count (i.e., the repeat-until loop in Line 4) and (2) afterwards we iteratively decrease the upper bound for bit-level costs until the SAT solver reports unsatisfiability for the first time (i.e., the repeat-until loop in Line 8). Finally the algorithm returns an adder graph with optimal bit-level costs for the minimum number of adders. As an additional benefit, we can use incremental solving in the second loop—if supported by the SAT solver. We use the trivial lower bound $N_{\text{LB}} = M$, the number of unique coefficients, since the SAT solver can quickly prove unsatisfiability in cases where a realization with M adders is infeasible. Note that other lower bounds (e.g., the one by Gustafsson [28]) can be used alternatively.

Algorithm 1 Computing an adder graph with optimal bit-level costs for the minimum adder count

```
Require: $C^{(0)}, \dots, C^{(M-1)}$                    ▷ Coefficients
Ensure: An adder graph $A$ with optimal bit-level costs
 1: $A \leftarrow \{\}$                       ▷ Initialize empty adder graph
 2: $N \leftarrow N_{\text{LB}}(C^{(0)}, \dots, C^{(M-1)})$          ▷ Lower bound
 3: $sat \leftarrow$ False
 4: repeat                                  ▷ Find optimal adder costs
 5:     $A, sat \leftarrow \texttt{solve}(C^{(0)}, \dots, C^{(M-1)}, N)$
 6:     if $\neg sat$ then
 7:         $N \leftarrow N + 1$               ▷ $N$ adders are insufficient
 8:     end if
 9: until $sat$
10: repeat                                  ▷ Find optimal bit costs
11:     $\hat{B} \leftarrow \texttt{compute\_bit\_costs}(A)$
12:     $\hat{A}, sat \leftarrow \texttt{solve}(C^{(0)}, \dots, C^{(M-1)}, N, \hat{B} - 1)$
13:     if $sat$ then
14:         $A \leftarrow \hat{A}$                    ▷ Found better solution
15:     end if
16: until $\neg sat$
17: return $A$
```

To simplify our algorithm, we only focus on bit operations exceeding the chosen input word size W_{in}. Hence, the actual number of bits is

$$\#\text{Bits} = N^* \cdot W_{\text{in}} + B^* \quad (1)$$

where N^* and B^* represent the optimal number of adders and the optimal number of (additional) bit operations reported by the solver.

In the example from Section II, our solver would report that it found a solution for zero additional bit operations. The computation of $5x$ requires 6 bit operations (0 bits more than the input word size) and the computation of $21x$ also requires 6 bit operations (also 0 bits more than the input word size).

In the following we describe how to solve the SCM problem using SAT (Section IV); extend it to MCM (Section V); extend it to signed fundamentals (Section VI); and finally extend it to bit-level costs (Section VII).

IV. SAT formulation for SCM

A SAT solver determines whether a Boolean expression can evaluate to true. If this is the case, the solver also gives a satisfying assignment of its variables. Otherwise, the solver performs an exhaustive search to show that such an assignment cannot exist. Usually, modern SAT solvers require the expression to be in conjunctive normal form (CNF). Such an expression consists of a conjunction of clauses; each clause is a disjunction of literals; and each literal is either a Boolean variable or its negation.

To decide whether there exists an SCM circuit for a given constant, maximum adder count and maximum bit-level cost, we have to convert that decision problem into a circuit with a given adder count N and bit-level cost $\hat{B}$, translate it to a single Boolean formula in conjunctive normal form (CNF) and test the CNF for satisfiability. To do so, every sub-circuit leads

TABLE III: Truth table for a MUX

d_0	d_1	s	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

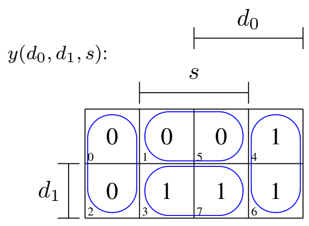


Fig. 2: K-map for a MUX

to clauses that have to be added to the Boolean expression. In the following, we provide a detailed example on how to model a multiplexer (MUX) as a first component within SAT.

Table III and Fig. 2 show the truth table and the Karnaugh map for a 2:1 MUX with data inputs d_0, d_1, select line s and output y. The MUX is fully characterized via the following four expressions (highlighted blue in Fig. 2):

$$(\neg s \land \neg d_0) \rightarrow \neg y \tag{2}$$
$$(s \land \neg d_1) \rightarrow \neg y \tag{3}$$
$$(\neg s \land d_0) \rightarrow y \tag{4}$$
$$(s \land d_1) \rightarrow y \tag{5}$$

Rearranging using the identity $a \rightarrow b = \neg a \lor b$, where $\rightarrow$ is the logical implication operator, yields cnf $= (s \lor d_0 \lor \neg y) \land (\neg s \lor d_1 \lor \neg y) \land (s \lor \neg d_0 \lor y) \land (\neg s \lor \neg d_1 \lor y)$. Passing this CNF to the SAT solver would result in the solver reporting *satisfiability*, with a satisfying assignment being, e.g., d_0, d_1, $\neg s$, y. Note that this is only one out of several satisfying assignments.

We use this approach to derive clauses for arbitrary logical operations.

A. Adder node circuit

Figure 3 shows the circuit that represents the i^{th} adder node that computes coefficient $c^{(i)}$. Its input MUX is connected to the outputs of all previous nodes and values for $\alpha^{(i)}$ and $\beta^{(i)}$ are determined by the SAT solver to select appropriate inputs. The left input, selected via $\alpha^{(i)}$, always gets shifted with its shift length determined by $\gamma^{(i)}$.

Bit $\varepsilon^{(i)}$ controls whether an addition or a subtraction is performed and the two MUXs, controlled by $\delta^{(i)}$, determine if $s^{(i)}$ or $r^{(i)}$ gets subtracted. The value for $\delta^{(i)}$ is therefore only relevant in case $\varepsilon^{(i)} = 1$. The two's complement conversion to model a subtraction is performed by the bitwise XOR gate, controlled by $\varepsilon^{(i)}$, and by setting the carry input of the adder equal to $\varepsilon^{(i)}$. Lastly, a right shift by $\zeta^{(i)}$ bits is performed, which is necessary to achieve the minimum number of adders for some constants (Section VIII).

The circuit that describes the overall optimization problem is built by using the adder circuit N times, i.e., creating the corresponding clauses for $i = 1 \ldots N$. The details of the clauses are described next.

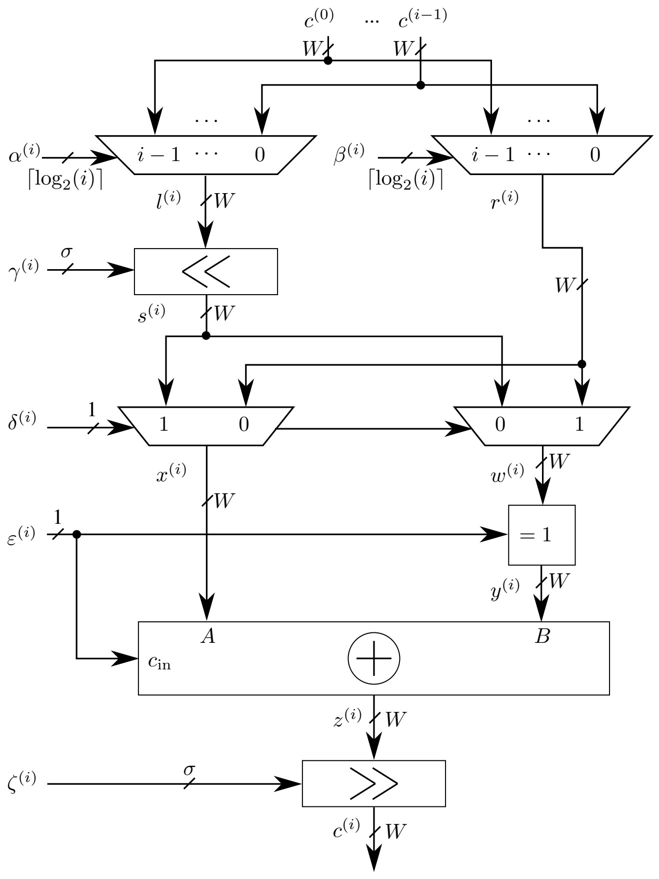


Fig. 3: Circuit description for one adder node

B. Clauses for basic operations

In the following, we define all clauses to represent the operations from Fig. 3. They directly follow from the truth tables of the respective operations and are minimized using Boolean algebra whenever applicable.

The 2:1 MUX with select input s, data inputs d_0, d_1 and output y is described by the following clauses:

$$\neg d_0 \lor s \lor y \tag{6}$$
$$d_0 \lor s \lor \neg y \tag{7}$$
$$\neg d_1 \lor \neg s \lor y \tag{8}$$
$$d_1 \lor \neg s \lor \neg y \tag{9}$$
$$\neg d_0 \lor \neg d_1 \lor y \tag{10}$$
$$d_0 \lor d_1 \lor \neg y \tag{11}$$

These are identical to the ones derived from (2)–(5) including two redundant clauses, which we add for improved performance during unit propagation [29]. We add (6)–(11) for all MUXs that appear in each adder node (Fig. 3). For example, when building clauses for the MUX with output $x^{(i)}$, we substitute $x_w^{(i)} = y$, $\delta^{(i)} = s$, $s_w^{(i)} = d_1$, $r_w^{(i)} = d_0$.

An XOR with inputs a and b and output y is modeled via:

$$a \vee b \vee \neg y \tag{12}$$

$$a \vee \neg b \vee y \tag{13}$$

$$\neg a \vee b \vee y \tag{14}$$

$$\neg a \vee \neg b \vee \neg y \tag{15}$$

The adder is realized as a ripple carry adder using full adders. A full adder with inputs a, b and c_{in} and output s (a 3-input XOR) is formulated as:

$$a \vee b \vee c_{\text{in}} \vee \neg s \tag{16}$$

$$a \vee b \vee \neg c_{\text{in}} \vee s \tag{17}$$

$$a \vee \neg b \vee c_{\text{in}} \vee s \tag{18}$$

$$\neg a \vee b \vee c_{\text{in}} \vee s \tag{19}$$

$$\neg a \vee \neg b \vee \neg c_{\text{in}} \vee s \tag{20}$$

$$\neg a \vee \neg b \vee c_{\text{in}} \vee \neg s \tag{21}$$

$$\neg a \vee b \vee \neg c_{\text{in}} \vee \neg s \tag{22}$$

$$a \vee \neg b \vee \neg c_{\text{in}} \vee \neg s \tag{23}$$

and the carry computation in the full adder with inputs a, b and c_{in} and output c_{out} is modeled using the following clauses:

$$\neg a \vee \neg b \vee c_{\text{out}} \tag{24}$$

$$a \vee b \vee \neg c_{\text{out}} \tag{25}$$

$$\neg a \vee \neg c_{\text{in}} \vee c_{\text{out}} \tag{26}$$

$$a \vee c_{\text{in}} \vee \neg c_{\text{out}} \tag{27}$$

$$\neg b \vee \neg c_{\text{in}} \vee c_{\text{out}} \tag{28}$$

$$b \vee c_{\text{in}} \vee \neg c_{\text{out}} \tag{29}$$

1) The input MUX: Each input MUX for node i has i data inputs. Therefore, node 1 does not need any input MUX because the left and the right inputs for node 1 are always node 0. Furthermore, an n:1 MUX is built out of a tree of 2:1 MUXs.

2) The left shifter: Fig. 4 shows our model for the left shifter. It is based on 2:1 MUXs, where stage w performs a left shift by 2^w bits if $\gamma_w = 1$.

The SAT solver is not allowed to produce overflows within the shifter. Hence, we must make sure that only zeros are shifted out on the MSB side. This means that we must account for four different types of MUXs:

1) a *regular* MUX
2) a MUX where $d_0 = 1$ and $s = 1$ is forbidden (to avoid shifting out ones on the MSB side)
3) a MUX with $d_1 = 0$ (to shift in zeros on the LSB side)
4) a MUX with $d_1 = 0$ where $d_0 = 1$ and $s = 1$ is forbidden (to shift in zeros on the LSB side and simultaneously avoid shifting out ones on the MSB side)

We implement a Type-1 MUX using (6)–(11). For a Type-2 MUX, we add the clause $\neg(d_0 \wedge s)$ to (6)–(11). Using a K-map minimization, we change (6) to

$$\neg d_0 \vee y \tag{30}$$

and (10) to

$$\neg d_0 \vee \neg s. \tag{31}$$

The remaining two MUX variants directly follow from setting $d_1 = 0$.

3) The adder: The adder is constructed as a ripple carry adder via (16)–(23) for the sum bits and via (24)–(29) for the carry bits. We create additional variables to represent the internal carry values and avoid overflows during addition by adding clauses

$$\neg c_{\text{out}} \vee \varepsilon^{(i)} \qquad \forall i \in [1,\, N] \tag{32}$$

$$c_{\text{out}} \vee \neg \varepsilon^{(i)} \qquad \forall i \in [1,\, N] \tag{33}$$

This ensures that subtractions always produce a carry output and additions never produce a carry output, which prohibits overflows since both numbers are guaranteed to be non-negative.

4) The right shifter: The right shifter works the same way as the left shifter, with the exception that the shift direction is reversed (i.e., zeros are shifted in at the MSB side and it is not allowed to shift out ones at the LSB side).

5) Input and output nodes: We force the input node (with index $i = 0$) to value 1 by setting $c_0^{(0)} = 1$ and $c_w^{(0)} = 0$ for $w \neq 0$ and we ensure that the output node (with index $i = N$) computes the requested constant by adding clause

$$\tilde{c}_w^{(i)} \qquad \text{with} \qquad \tilde{c}_w^{(i)} = \begin{cases} \neg c_w^{(i)} & \text{if } C_w = 0 \\ c_w^{(i)} & \text{if } C_w = 1 \end{cases} \tag{34}$$

for each bit $w \in [0,\, W-1]$.

6) Accelerating unsat proofs: The solver is required to prove unsatisfiability for all values for N that are too small to express the target constant. As it was proven by Dempster and Macleod [16], it suffices to only consider odd fundamentals to guarantee minimum adder count. We therefore propose to add clause

$$c_0^{(i)} \qquad \forall i \in [1,\, N] \tag{35}$$

to the formulation. This sets each LSB to true, which forces all fundamentals to odd values. Even through this clause is not required for correctly modeling the SCM problem, we found that it helps speeding up the solving process for unsatisfiable instances. As an additional benefit, it also simplifies the extension to bit-level costs (Section VII).

V. Extension to MCM

The extension to MCM is straightforward. Instead of ensuring that the SCM constant of interest gets computed by the last node, we must make sure that each MCM constant gets computed by at least one node. To do so, we use variables $t^{(i,m)}$ and make sure that the solver is only able to set $t^{(i,m)} = 1$ whenever $c^{(i)} = C^{(m)}$ by forcing the relationship

$$t^{(i,m)} \rightarrow (c^{(i)} = C^{(m)}) \qquad \forall i \in [1,\, N], m \in [1,\, M] \tag{36}$$

using clauses

$$\neg t^{(i,m)} \vee \tilde{c}_w^{(i,m)} \qquad \forall i \in [1,\, N], m \in [1,\, M] \tag{37}$$

with $\tilde{c}_w^{(i)}$ defined as

$$\tilde{c}_w^{(i,m)} = \begin{cases} \neg c_w^{(i)} & \text{if } C_w^{(m)} = 0 \\ c_w^{(i)} & \text{if } C_w^{(m)} = 1 \end{cases} \tag{38}$$

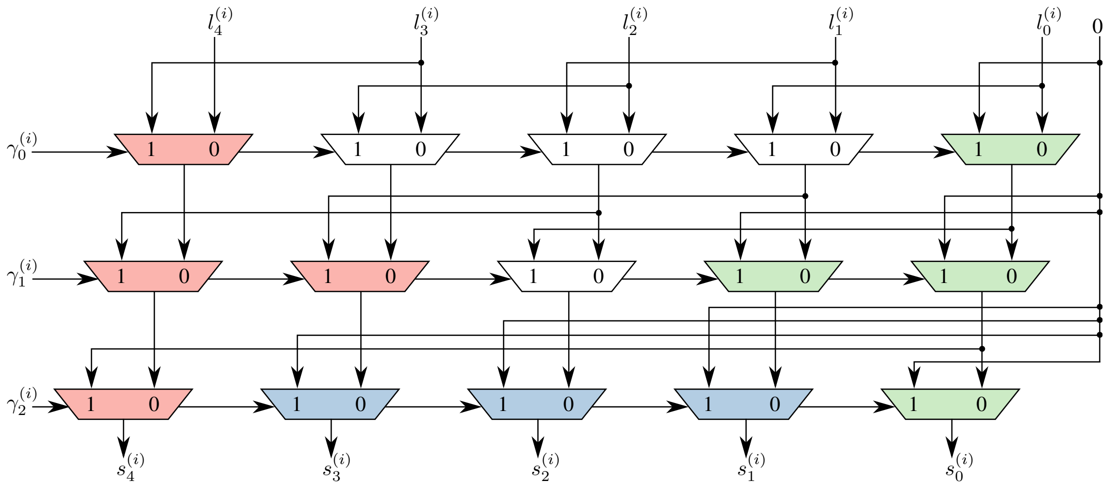


Fig. 4: Left shifter circuit for $W = 5$, built from 2:1 MUXs; white: Type-1, red: Type-2, green: Type-3, blue: Type-4

Finally, we make sure that each MCM constant gets computed by at least one node via clause

$$\bigvee_{i=1}^{N} t^{(i,m)} \qquad \forall m \in [1,\, M]. \tag{39}$$

VI. Extension to signed fundamentals

As shown by Gustafsson et al. [19], negative fundamentals are sometimes cheaper to realize than their positive counterparts, because the operation $x - (y \ll s)$ consumes less resources than $(x \ll s) - y$. Note that this has no impact on the optimal number of adders but the extension to signed fundamentals prepares for the bit-level cost model described in Section VII.

Being bit-level-aware has the advantage that we can also directly compute negative output fundamentals if their computation is cheaper than computing the positive counterpart and let the solver choose which one of them to realize. In MCM problems, where both the positive and the negative coefficient are requested, it is natural to let the solver decide which one to compute. In cases where a positive coefficient is requested but the sign of an output fundamental does not matter[2], we can also let the solver decide the sign.

To support signed fundamentals, we must represent all numbers as two's complement numbers, which requires increasing the SAT solver's internal word size by 1 (i.e., $W = \max_n \lceil \log_2 C_n \rceil + 2$). Furthermore, we must change the rules for overflow protection in the adder and the rules regarding the MSB within the shifters as described in the following.

A. Signed overflow protection in the adder

Overflows in two's complement computations depend on the sign bits of both inputs, on the output sign bit and also on the operation performed (addition or subtraction). Overflows occur in the following four cases:

[2]This might be the case if the MCM circuit is embedded in a larger system, such as a digital filter, where computing a coefficient with the "wrong" sign can be accounted for without additional hardware costs by changing, e.g., a subsequent adder to a subtractor.

1) $z = x + y$ with $x > 0, y > 0, z < 0$,
2) $z = x + y$ with $x < 0, y < 0, z > 0$,
3) $z = x - y$ with $x > 0, y < 0, z < 0$, and
4) $z = x - y$ with $x < 0, y > 0, z > 0$.

We prohibit these cases using clauses

$$\varepsilon^{(i)} \vee x_{W-1}^{(i)} \vee u_{W-1}^{(i)} \vee \neg a_{W-1}^{(i)}, \tag{40}$$

$$\varepsilon^{(i)} \vee \neg x_{W-1}^{(i)} \vee \neg u_{W-1}^{(i)} \vee z_{W-1}^{(i)}, \tag{41}$$

$$\neg\varepsilon^{(i)} \vee x_{W-1}^{(i)} \vee \neg u_{W-1}^{(i)} \vee \neg z_{W-1}^{(i)}, \text{ and} \tag{42}$$

$$\neg\varepsilon^{(i)} \vee \neg x_{W-1}^{(i)} \vee u_{W-1}^{(i)} \vee z_{W-1}^{(i)} \tag{43}$$

for all $i \in [1,\, N]$

B. Signed MSB rules in the shifters

In the left shifter, we are allowed to shift out zeros on the MSB side for positive inputs and ones for negative inputs. This affects Type-2 and Type-4 MUXs. Additionally, we must make sure that the sign bit before and after shifting remains the same via clauses

$$\neg l_{W-1}^{(i)} \vee s_{W-1}^{(i)} \qquad \text{and} \qquad l_{W-1}^{(i)} \vee \neg s_{W-1}^{(i)} \tag{44}$$

for all $i \in [1,\, N]$

To account for signed fundamentals in the left shifter, Type-2 MUXs change to Type-1 MUXs and Type-4 MUXs change to Type-3 MUXs and we add clauses

$$\neg s \vee \neg \text{sign}(d_0) \vee d_{1,w} \tag{45}$$

$$\neg s \vee \text{sign}(d_0) \vee \neg d_{1,w} \tag{46}$$

for MUXs at position $w < W - 1$ with s being the select input, $\text{sign}(d_0)$ being the sign bit of the data input for $s = 0$ and $d_{1,w}$ being bit w of the data input for $s = 1$. We also add clauses

$$\neg s \vee \neg \text{sign}(d_0) \vee d_{0,w} \tag{47}$$

$$\neg s \vee \text{sign}(d_0) \vee \neg d_{0,w} \tag{48}$$

for MUXs at position $w = W - 1$. These additional clauses ensure that only bits equal to the sign bit are shifted out.

Handling the right shifter is simpler as overflows cannot occur. We only need to shift in the sign bit instead of zeros on the MSB side, changing Type-3 MUXs to Type-1 and Type-4 MUXs to Type-2. For all former Type-3 and Type-4 MUXs, the sign bit is connected to d_1.

VII. Extension to bit-level costs

Minimizing the number of adders is a reasonable high-level objective for the general SCM/MCM problem but the choice of non-output fundamentals and the choice of operations ($+$, $-$ and shift) influences the bit-widths of the adders, as seen in Section II. Hence, we extend our SAT formulation to also support an upper limit on the number of available bit operations. This way, we can first generate a solution for the minimum amount of adders and afterwards refine that solution by optimizing the bit-widths of the adders.

Our SAT formulation relies on the following assumptions:

1) As mentioned in the Introduction, adders are implemented as ripple carry adders.
2) Inputs to the SCM/MCM circuit are represented as signed integers in two's complement.
3) The input word size is larger than the maximum shift (otherwise we would not need any adders for large shifts because the input operands do not overlap).
4) An n bit adder and an n bit subtractor have equal costs (i.e., the negations for subtractions can be implemented without additional costs).
5) $c^{(i)}$ is odd.
6) Input operands for additions/subtractions always overlap.

These assumptions result in the following conclusions:

1) The adder word size of a node is $W_{\text{in}} + \lceil \log_2(|z^{(i)}|) \rceil$ where W_{in} is the input word size.
2) For additions, the number of bit operations is $W_{\text{in}} + \lceil \log_2(|z^{(i)}|) \rceil - \gamma^{(i)}$.
3) For $x - (y \ll s)$, we can exploit that the shifted result gets negated, turning the trailing zeros into ones, and the carry input for the two's complement conversion turns the ones into zeros again and we account for the carry input at bit position s.
4) For $(x \ll s) - y$, we cannot save bits on the LSB side because they are needed to perform the inversion of y. The number of bits is therefore equal to $W_{\text{in}} + \lceil \log_2(|x^{(i)}|) \rceil$.
5) If a node has inputs with the same sign, the sign bit can be copied from one of those inputs, potentially saving one bit per adder.

Under these assumptions, our bit-level cost model equally weighs full adders, half adders and inverters. This means that we can accurately model the number of LUTs on an FPGA, since all circuit elements (half adder, full adder, inverter) consume exactly one LUT.

For ASICs, proposed algorithm can be seen as a heuristic: ASIC area depends on bit-width and, beyond that, on effort per bit position. Our solution does not account for bit position costs as we model every bit operation as a full adder. In concrete cases, individual bit operations might be reduced to a half adder if one of the inputs is missing, or even to an XOR if only the sum is required (in both cases with optional inversions in case of a subtraction). Therefore, our model will fail to identify solutions with more bit operations but less chip area. However, when compared to previous solutions that neglect adder bit widths, our method consistently reduces the number of bit operations significantly (as shown later bin Fig. 10). On average, this reduction should far outweigh any potential increase in cost for some individual bits.

Already noted by Garcia et al. [11], an adder can be completely omitted if its input operands do not overlap (i.e., Assumption 6 is violated). This can occur only if the pre-addition shift length exceeds the second adder input's word size. Like Garcia et al. we do not consider this case in our work, since it hardly ever occurs in practice. The proposed algorithm overestimates bit-level costs if Assumption 6 does not hold.

Depending on the operation performed, three cases can affect the number of saved bits on the LSB side:

1) $(x \ll s) + y$ or $x + (y \ll s)$;
2) $x - (y \ll s)$;
3) $(x \ll s) - y$.

For case 1 & 2 we calculate the number of bits for node i, $B^{(i)}$, via

$$B^{(i)} = W_{\text{in}} + \left\lceil \log_2(|z^{(i)}|) \right\rceil - \gamma^{(i)} - m^{(i)}. \tag{49}$$

For case 3 the number of bits is

$$B^{(i)} = W_{\text{in}} + \left\lceil \log_2(|z^{(i)}|) \right\rceil - m^{(i)}. \tag{50}$$

Here, $m^{(i)}$ is a variable that denotes whether the MSB can be copied from one of the node's inputs. When restricting the solver to use only positive coefficients, $m^{(i)}$ is always 1. Since W_{in} is a constant and the same for all adder nodes, we do not consider W_{in} within our SAT formulation.

The synthesis tool might sometimes find opportunities to reduce bit-level costs even further, which our algorithm cannot account for (e.g., in cases where $l^{(i)} = r^{(i)}$). Note that this may only *reduce* the hardware requirements of the resulting circuit.

A. Word size increase due to the computed constant

To calculate $\lceil \log_2(|z^{(i)}|) \rceil$, we compute an intermediate result $a^{(i)} = |z^{(i)}|$ by modeling the circuit shown in Fig. 5. The MUXs select $-z^{(i)}$ if $z^{(i)} < 0$ (i.e., when its sign bit is set to 1) and $z^{(i)}$ if $z^{(i)} \geq 0$ (i.e., when its sign bit is set to 0).

We model Algorithm 2 within SAT to calculate $\lceil \log_2(a^{(i)}) \rceil$. It works by first initializing two variables: an integer v that will hold the result, and a binary carry variable c that denotes whether a leading zero was already found in a previous iteration. Starting at the MSB, the for-loop iterates through the bit vector that represents the adder result and adds one to v whenever the current or a previous bit is set to 1. After the last iteration (for $w = 0$), v holds the word size of $a^{(i)}$.

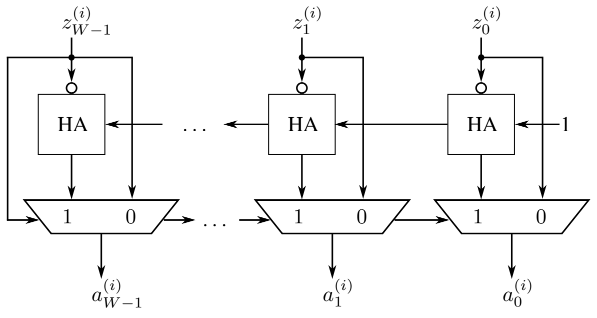


Fig. 5: Circuit to compute $a^{(i)} = |z^{(i)}|$

Algorithm 2 Computing $\lceil \log_2(a^{(i)}) \rceil$

```
Require: $a^{(i)}$
Ensure: $v^{(i)} = \lceil \log_2(a^{(i)}) \rceil$
 1: $v \leftarrow 0$                ▷ Integer variable that will hold the result
 2: $c \leftarrow 0$                ▷ Binary carry variable
 3: for $w \leftarrow W-1 \ldots 0$ do
 4:     $t_1 \leftarrow c \vee a_w^{(i)}$        ▷ $t_1$ is 1 bit
 5:     $t_2 \leftarrow v \,\&\, c$        ▷ Bitwise $\wedge$ relation; $t_2$ is an integer
 6:     $v \leftarrow t_1 + t_2$
 7:     $c \leftarrow t_1$
 8: end for
 9: $v^{(i)} \leftarrow v$
10: return $v^{(i)}$
```

B. Cutting LSBs

To distinguish between (49) and (50), we introduce new variables $g^{(i)}$ that denote the number of LUTs saved on the LSB side. Therefore, $g^{(i)} = 0$ for $(x \ll s) - y$ and $g^{(i)} = \gamma^{(i)}$ in all other cases.

We enforce this constraint for each bit w in $g^{(i)}$ by adding clauses

$$\gamma_w^{(i)} \vee \neg g_w^{(i)} \tag{51}$$
$$\neg\delta^{(i)} \vee \neg\varepsilon^{(i)} \vee \neg g_w^{(i)} \tag{52}$$
$$\delta^{(i)} \vee \neg\gamma_w^{(i)} \vee g_w^{(i)} \tag{53}$$
$$\varepsilon^{(i)} \vee \neg\gamma_w^{(i)} \vee g_w^{(i)} \tag{54}$$

for all $i \in [1,\, N]$. Clause (51) ensures that the solver never cuts more LSBs than the shift size. We use (52) to handle the case $(x \ll s) - y$ where we cannot save bits on the LSB side. The remaining two cases are modeled by (53) and (54), where the solver is allowed to cut bits on the LSB side.

C. Copying the MSB

The MSB for a given node does not have to be explicitly computed if at least one of its inputs has the same sign. This might reduce the number of full adders by one per adder. We therefore add variables $m_w^{(i)}$ to represent whether the sign bit of a node's output can be copied from one of its inputs and add clauses

$$u_{W-1}^{(i)} \vee z_{W-1}^{(i)} \vee m_w^{(i)}, \tag{55}$$
$$\neg u_{W-1}^{(i)} \vee x_{W-1}^{(i)} \vee m_w^{(i)}, \tag{56}$$
$$\neg x_{W-1}^{(i)} \vee \neg z_{W-1}^{(i)} \vee m_w^{(i)}, \tag{57}$$
$$\neg x_{W-1}^{(i)} \vee \neg u_{W-1}^{(i)} \vee z_{W-1}^{(i)} \vee \neg m_w^{(i)}, \text{ and} \tag{58}$$
$$x_{W-1}^{(i)} \vee u_{W-1}^{(i)} \vee \neg z_{W-1}^{(i)} \vee \neg m_w^{(i)} \tag{59}$$

for all $i \in [1,\, N]$. This ensures that the SAT solver is only allowed to copy the sign bit for a node that computes a negative number if at least one of its inputs is also negative and the sign bit for positive fundamentals can only be copied if at least one input is positive. If the SAT solver is restricted to use only positive fundamentals, $m^{(i)}$ should be set to true via clause

$$m^{(i)} \qquad \forall i \in [1,\, N] \tag{60}$$

There are no sign bits in case of an unsigned input. Nevertheless, we can omit the explicit computation of the MSB in a given adder node, whenever the output word size of that adder is exactly one bit larger than its largest input word size. The derivation of the corresponding clauses for unsigned inputs is straightforward and we omit it here since we focus on signed inputs due to their practical relevance for digital filter implementations.

D. Calculating final bit-level costs

Bit-level costs can, in general, be negative within the SAT formulation because they describe the offset w.r.t. the input word size (B^* in (1)), which requires computing bit-level costs using two's complement numbers. We therefore reorder the computation as

$$B = \Sigma_{\mathrm{v}} - (\Sigma_{\mathrm{g}} + \Sigma_{\mathrm{m}})$$

with

$$\Sigma_{\mathrm{v}} = \sum_{i=1}^{N} v^{(i)}, \quad \Sigma_{\mathrm{g}} = \sum_{i=1}^{N} g^{(i)}, \quad \Sigma_{\mathrm{m}} = \sum_{i=1}^{N} m^{(i)}. \tag{61}$$

This allows us to model the computation of Σ_{v}, Σ_{g}, and Σ_{m} and also the sum $\Sigma_{\mathrm{g}} + \Sigma_{\mathrm{m}}$ using ripple-carry adders and unsigned arithmetic.

E. Comparing bit-level costs to maximum costs

Finally, the SAT solver must compare bit-level costs for the current adder graph B to an upper limit $\hat{B}$ to be able to decide whether $\hat{B}$ is satisfiable. This is done in Algorithm 3.

The for loop iterates through all bits of B and generates clauses for two variables: ok_w and c. Similar to Algorithm 2, we use c as a carry variable which is set to true as long as the MSBs of B and $\hat{B}$ are equal. The ok_w bit denotes whether $B \leq \hat{B}$ is partially satisfied for all bits at positions greater than or equal to w. $B \leq \hat{B}$ is only satisfied if and only if all ok_w bits are true. We therefore add clauses

$$\mathrm{ok}_w \qquad \forall w \in [0,\, W_{\mathrm{B}} - 1]. \tag{62}$$

Algorithm 3 Checking whether $B \leq \hat{B}$

```
Require: $B$, $\hat{B}, W_\mathrm{B}$
 1: for $w \leftarrow \lceil \log_2 W_\mathrm{B} \rceil - 1 \ldots 0$ do
 2:     if $w = W_\mathrm{B} - 1$ then
 3:         if $\hat{B}_w = 1$ then
 4:             $\mathrm{ok}_w \leftarrow B_w$
 5:             $c \leftarrow L_w$
 6:         else
 7:             $\mathrm{ok}_w \leftarrow \mathrm{true}$
 8:             $c \leftarrow \neg B_w$
 9:         end if
10:     else
11:         if $\hat{B}_w = 1$ then
12:             $\mathrm{ok}_w \leftarrow \mathrm{ok}_{w+1} \vee c$
13:             $c \leftarrow c \wedge B_w$
14:         else
15:             $\mathrm{ok}_w \leftarrow (\neg c \wedge \mathrm{ok}_{w+1}) \vee (c \wedge \neg B_w)$
16:             $c \leftarrow c \wedge \neg B_w$
17:         end if
18:     end if
19: end for
20: return ok
```

All if-else conditions in Algorithm 3 can be evaluated "offline" (i.e., while setting up the SAT fomulation) because $\hat{B}$ is a constant. The if condition in Line 3 handles the sign bit separately and depending on the current bit $\hat{B}$, we must add clauses that model the respective Boolean expressions.

VIII. Experiments

We conduct experiments on an AMD EPYC 7443P CPU @ 3.77 GHz using CaDiCaL [30] to solve SAT instances and Gurobi [31] for ILP instances, both limited to one thread each. In our initial experiments, we also used Z3 [32] and Glucose [33] as SAT solvers but found that CaDiCaL always had the lowest CPU time among all solvers. We therefore only report results obtained via CaDiCaL. Results for the Branch and Bound (BnB) algorithm [5] are obtained using a highly-optimized C++ implementation by Leothaud [34] instead of the original Matlab implementation used by Aksoy et al. [5].

For all experiments, we reproduce results for the ILP-based approaches by Kumm [4] and Garcia and Volkova [12] and for BnB by Aksoy et al. [5] on our machine. For the SAT-based SCM algorithm by Lagoon and Metodi [7], however, we were not able to reproduce results and instead report experimental results given in the paper.

Our SCM benchmarks are based on (i) all 20-bit numbers that can be realized with up to five adders and (ii) tests done by Lagoon and Metodi to evaluate the practicability of their SAT-based SCM algorithm. For MCM we focus on image processing filter kernels that are also used to benchmark ILP-based MCM algorithms [4], [11].

A. Objective: minimize the adder count

As our first experiment, we compare our algorithm to the approach by Lagoon and Metodi [7] which aims at solving the SCM problem for the minimum number of adders. For completeness, we also report CPU times for BnB [5] and for ILP [4]. Lagoon and Metodi report CPU times for their algorithm to determine that there cannot exist an implementation for certain constants with a certain number of adders. Results are summarized in Table IV. Our algorithm is able to reproduce all results. Although CPU time for our algorithm is lower for all target constants compared to Lagoon and Metodi's approach, they used another SAT solver and they do not report information about the CPUs used. Unfortunately, ILP and BnB do not return a solution for the two largest constants due to either running out of memory after one week of computation time or numerical errors in the ILP solver.

Secondly, we evaluate the importance of the post-add shift operation for SCM. So far, the necessity for right shifts in SCM have not been evaluated before. Fig. 6 shows the number of constants, representable with max. five adders and max. 20 bits, that need this shift to achieve optimal adder costs. Although that number is small—only 0.065 %—any optimal SCM algorithm must incorporate that shift to guarantee optimal adder costs in the general case. The smallest target constant that needs this shift is $C = 39,757$.

For the MCM experiments (Table V), we compared our algorithm's performance with a recent ILP-based approach [4], using the same benchmark instances. Our algorithm consistently outperforms the ILP-based approach in terms of runtime, completing all test instances within five seconds. In contrast, the ILP-based approach times out for one instance, resulting in a significantly increased runtime of twelve hours, yielding a non-optimal solution. Additionally, Aksoy et al.'s Branch and Bound (BnB) approach [5] quickly solves all instances for optimal adder costs in under one second but lacks support for bit-level optimization. Furthermore, BnB solving times do not scale well for large coefficients (Table IV).

TABLE IV: Proving that there does not exist an implementation with exactly N adders for constant C; *oom*: out of memory (512 GB); *err*: numerical errors while solving

		CPU time [s]			
N	C	SAT [7]	BnB [5]	ILP [4]	prop.
1	11	0	0	0	0
2	43	0	0	0	0
3	683	0	0	1	0
4	14,709	15	1	218	2
5	699,829	2 111	683	1154	375
5	171,398,451	1 120	*oom*	*err*	220
6	171,398,453	520 751	*oom*	*err*	227 530

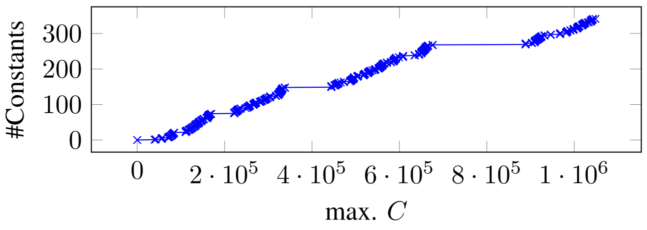
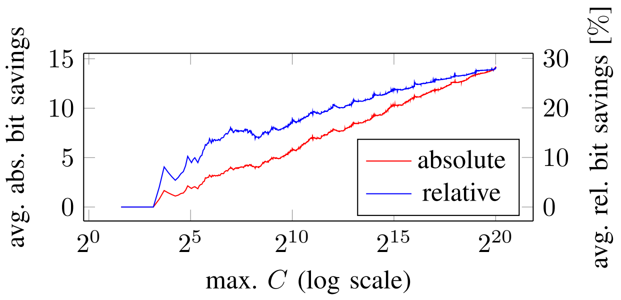


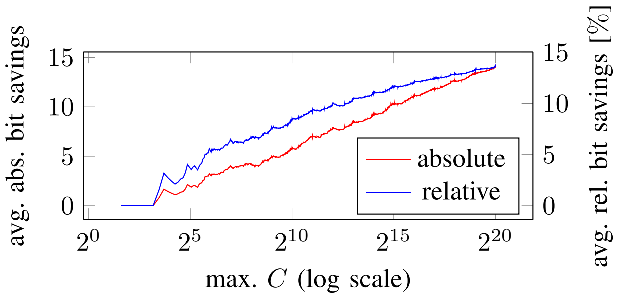
Fig. 6: The number of constants which require the post-add left shift to achieve the optimal number of adders

TABLE V: MCM results with objective min. # add

type	size	W	M	# add		CPU time [s]	
				ILP [4]	prop.	ILP [4]	prop.
gaussian	3×3	8	3	4	4	0	0
gaussian	5×5	12	3	5	5	18	1
laplacian	3×3	8	3	3	3	0	0
unsharp	3×3	8	3	4	4	0	0
unsharp	3×3	12	3	5	5	2	0
lowpass	5×5	8	5	6	6	2	0
lowpass	9×9	10	12	12	12	191	0
lowpass	15×15	12	25	31	**25**	43 867	4
highpass	5×5	8	4	4	4	0	0
highpass	9×9	10	5	5	5	0	0
highpass	15×15	12	12	12	12	1	0



(a) $W_{\text{in}} = 8$



(b) $W_{\text{in}} = 24$

Fig. 7: SCM bit savings

B. *Objective: minimize the number of bits (pre-synthesis)*

Figure 7 shows bit savings when running our algorithm for bit-level cost minimization on 2,125 randomly chosen coefficients from the SCM benchmark set[3] evaluated for an input word size of $W_{\text{in}} = 8$ and $W_{\text{in}} = 24$. We see that average cost savings approach $28\,\%$ for $W_{\text{in}} = 8$ and $15\,\%$ for $W_{\text{in}} = 24$ compared to solutions generated with our approach with optimal adder count only. This clearly shows the necessity of bit-level optimizations for SCM circuits.

For bit-level optimized MCM benchmarks, we compare our approach with bit-level optimizations to Garcia and Volkova's ILP formulation [12]. Their approach assumes unsigned inputs, but it can also be applied to signed inputs without guaranteeing optimality in terms of bit-level costs. For a fair comparison, we conduct two evaluations: one for signed inputs and one for unsigned ones; results can be found in Table VI. Notably, our algorithm outperforms the ILP approach for signed inputs in all but two test instances, where costs are equal. This difference arises from their approach not accounting for signed fundamentals, enabling our approach to reduce bit-level costs in nearly all test instances.

Our algorithm can handle unsigned inputs in two different ways. The first one is to increase the word size by one, pad the input with a leading zero on the MSB side and model a circuit for signed arithmetic (denoted as prop$^{(s)}$ in Table VI). The second way is to prohibit negative fundamentals and model a circuit for unsigned arithmetic (denoted as prop$^{(u)}$). It can be seen that in several instances, increasing the word size by one bit and using signed fundamentals results in even lower bit-level costs than the corresponding circuit for optimal unsigned arithmetic. In two instances, the ILP approach is able to reduce hardware costs (pre-synthesis) by one and two bit operations, respectively, compared to the proposed one. In all other instances, hardware costs obtained by our approach are either lower or equal. Differences between prop$^{(u)}$ are due to the ILP model incorporating rules to avoid computing the MSB for unsigned inputs, which is not supported by the proposed SAT formulation since we focus on signed input data due to the practical relevance.

C. *Objective: minimize the LUT count (post-place & route)*

Figure 8 shows the optimization potential due to incorporating bit-level costs by comparing synthesized MCM circuits (for signed inputs and $W_{\text{in}} = 8$) obtained via our proposed approach, the ILP formulation [12], and the BnB algorithm [5]. Here, BnB serves as a baseline, because it only generates a solution for the optimal adder costs without incorporating a detailed bit-level cost model.

LUT savings compared to the baseline with only optimal adder count (BnB) range from $1.9\,\%$ (Flt. 4, unsharp 3×3 with $W = 12$) up to $24.5\,\%$ (Flt. 3, unsharp 3×3 with $W = 8$). For some filters, Vivado is able to cut a few LUTs during logic synthesis, which neither approach accounts for. This leads to ILP beating our approach by 2 and 1 LUTs for Flt. 0 and Flt. 6, respectively. Averaged over all filters, we calculate LUT savings of $17.0\,\%$ for our proposed approach and $9.7\,\%$ for ILP. Total LUT savings for all filters are $15.4\,\%$ for ours and $7.9\,\%$ for ILP.

D. *Design space exploration (pre-synthesis)*

The last set of experiments enumerates the whole design space for some selected SCM/MCM problems to evaluate the potential of using bit-level metrics and signed fundamentals in the optimization. For that, we add a clause each time the SAT solver finds a solution for a given adder count to prohibit that solution and start solving again with this additional clause. We repeat this process until the solver reports unsatisfiability, which means that all solutions were found[4].

Figure 9 shows histograms of all possible adder graphs that lead to $C = 14,709$ with optimal adder count and various

[3]For each bit widths up to 20 bits we chose up to 50 random coefficients per optimal adder count.

[4]for the optimal adder count N^*, max. shift $S = \max_m \lceil \log_2 C^{(m)} \rceil$ and max. coefficient word size $W = S + 1$

TABLE VI: MCM results with objective min. # bits for $W_{\text{in}} = 8$; prop.$^{(u)}$ is our approach limited to only positive fundamentals and unsigned arithmetic; prop.$^{(s)}$ is our approach using signed fundamentals with signed arithmetic, using $W_{\text{in}} = 9$

					signed inputs				unsigned inputs					
					# bits		CPU time [s]		# bits			CPU time [s]		
Flt.	type	size	W	M	ILP [12]	prop.	ILP [12]	prop.	ILP [12]	prop.$^{(u)}$	prop.$^{(s)}$	ILP [12]	prop.$^{(u)}$	prop.$^{(s)}$
0	gaussian	3×3	8	3	39	39	3	0	**40**	41	43	4	0	0
1	gaussian	5×5	12	3	55	**52**	635	3	57	57	57	635	3	3
2	laplacian	3×3	8	3	30	**24**	0	0	31	31	**27**	0	0	0
3	unsharp	3×3	8	3	32	**30**	1	0	**32**	34	34	1	0	0
4	unsharp	3×3	12	3	47	**40**	143	2	49	49	**45**	143	1	2
5	lowpass	5×5	8	5	58	58	241	1	60	60	64	241	0	1
6	lowpass	9×9	10	12	121	**117**	7200	7200	130	**128**	129	7200	7200	7200
7	lowpass	15×15	12	25	–	**245**	7200	7200	–	**263**	270	7200	7200	7200
8	highpass	5×5	8	4	38	**29**	1	0	39	41	**33**	1	0	0
9	highpass	9×9	10	5	46	**36**	5	0	47	47	**41**	5	0	0
10	highpass	15×15	12	12	104	**87**	7200	7200	105	111	**99**	7200	7200	7200

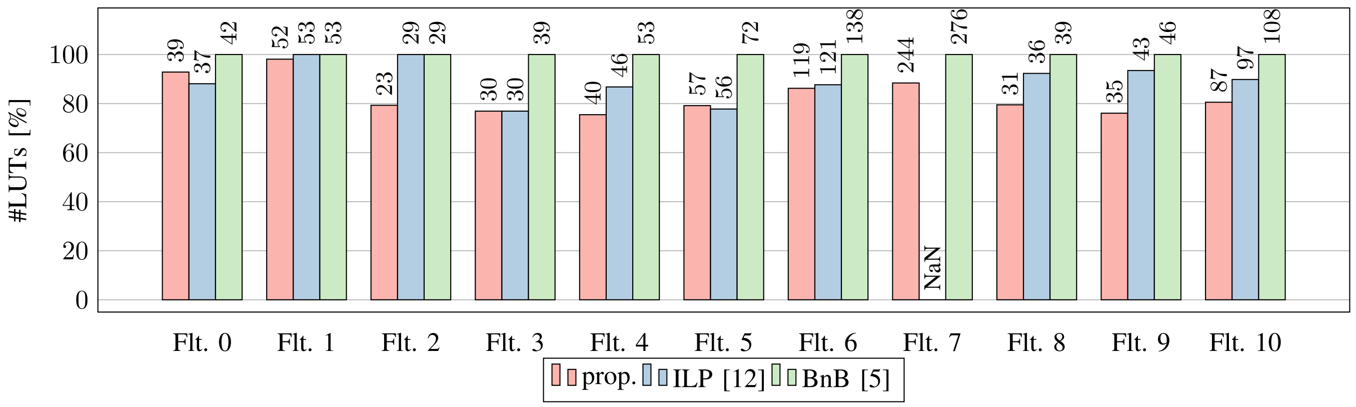
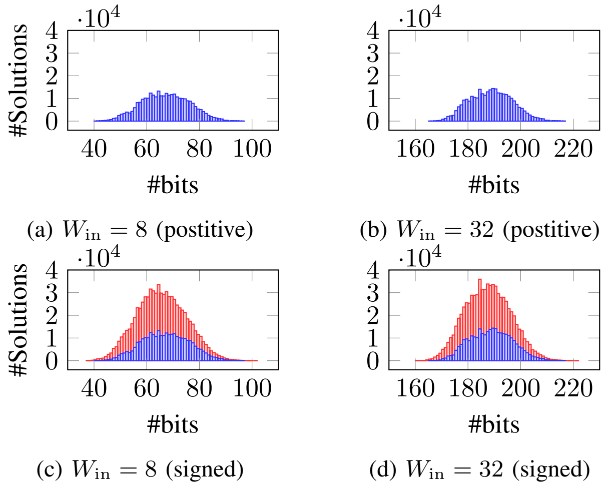


Fig. 8: LUT costs after Place & Route for signed inputs and $W_{\text{in}} = 8$ normed to results obtained via BnB [5]. Numbers above bars give absolute LUT counts (note that ILP is unable to compute a solution for Flt. 7). Savings averaged over all instances are $17.0\,\%$ for our proposed approach and $9.7\,\%$ for ILP. Absolute LUT savings for all filter instances are $15.4\,\%$ for our proposed approach and $7.9\,\%$ for ILP.

word sizes. We see a remarkable optimization potential for $W_{\text{in}} = 8$. In the optimal case for positive fundamentals, we can reach an implementation with only 40 bit operations whereas the worst-case SCM circuit needs 96 bit operations, even though both implementations have the same adder count $N = 5$. Using signed fundamentals, the optimal solution now needs 37 bits and the circuit with highest costs utilizes 101 bits. As seen in Fig. 9d, we still have a difference of approx. 60 bits between the optimal and worst cases for $W_{\text{in}} = 32$. Considering that a non-bit-aware SCM/MCM algorithm arrives at any of these implementations, only minimizing the adder count does not seem to be a reasonable optimization goal in hardware design; especially for low word sizes.

Digital filters (FIR and IIR) are examples, where the sign of the output fundamental in the MCM circuit is irrelevant because it can be compensated by changing a subsequent adder to a subtracter or vice versa. We therefore enumerate all possible solutions for MCM with the following settings:

1) Positive output and non-output fundamentals
2) Signed non-output fundamentals; positive output fundamentals in the filter description are realized as positive output fundamentals; negative filter coefficients can be realized as positive or negative output fundamentals



(a) $W_{\text{in}} = 8$ (postive)
(b) $W_{\text{in}} = 32$ (postive)
(c) $W_{\text{in}} = 8$ (signed)
(d) $W_{\text{in}} = 32$ (signed)

Fig. 9: All solutions with min. #adders for $C = 14{,}709$ and different word sizes for positive (blue) and signed fundamentals (red)

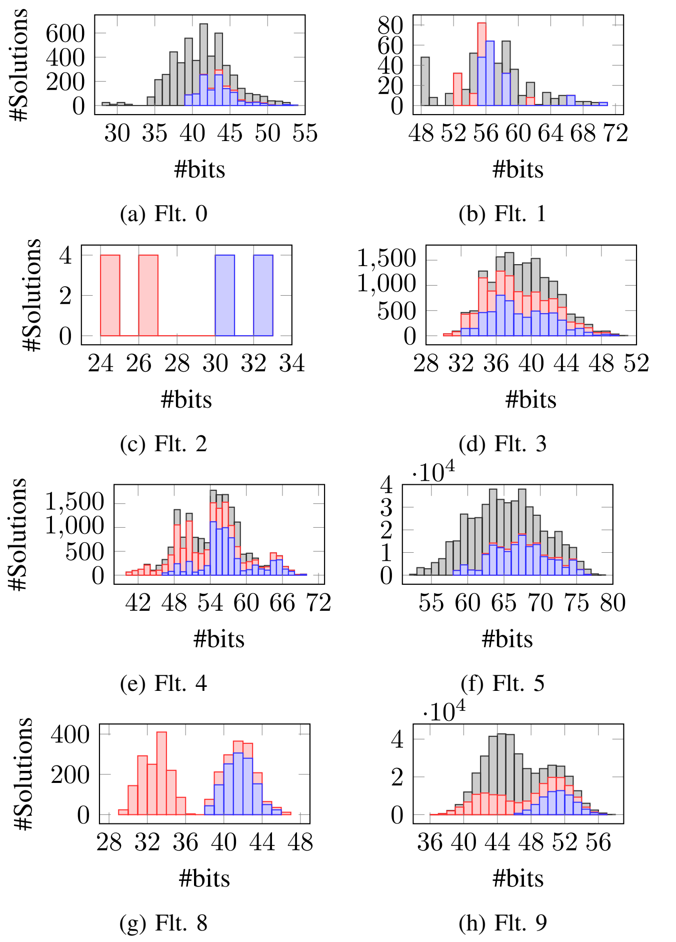


Fig. 10: All solutions with min. #adders for MCM instances and $W_{\text{in}} = 8$ for positive (blue); signed fundamentals with negative coefficients only allowed for non-output fundamentals and negative requested output fundamentals (red); and signed fundamentals with negative coefficients allowed for all fundamentals (grey)

3) *All* fundamentals are allowed to be negative (independent of the signs in the filter specification).

Design space exploration results for MCM are depicted in Fig. 10. A general observation is that the design space grows and the optimum solution regarding the number of bit operations is improved when allowing signed fundamentals. For some filters it is noteworthy that allowing any signed fundamentals at all allows for a completely new set of solutions with a lower number of bit operations (e.g., Flt. 2). Another interesting observation is that allowing the SAT solver to choose the sign of any output fundamental can decrease the optimum number of bit operations by a significant amount (e.g., approx. $25\,\%$ for Flt. 0). These results clearly demonstrate that the restriction to strictly positive fundamentals should be lifted whenever possible to decrease hardware requirements of digital filter implementations.

IX. Conclusion

In this paper we present a SAT based optimization algorithm for the SCM and MCM problems. Given enough computation time, our algorithm is able to compute a provably optimal MCM circuit for the minimum number of adders and the minimum number of bit-level costs (i.e., LUTs on an FPGA).

By comparing our work to a state-of-the-art algorithm for optimal SCM [7], we show that our formulation is competitive by proving optimality even for the "hardest" benchmarks (i.e., the smallest number that cannot be represented anymore with six adders: $171, 398, 453$). We also show that the post-adder right shifter is needed to guarantee minimal adder costs for SCM, albeit being necessary for only $0.065\,\%$ of all coefficients with up to 20 bits and max. five adders. For that same set of numbers, we are able to reduce bit-level costs by $28\,\%$ on average for an input word size of eight bits compared to solutions with optimal adder count.

By comparing our approach to a recent, state-of-the-art ILP based approach for adder-optimal MCM [4], we show that our algorithm scales better to large problem sizes by having a significantly reduced runtime that also leads to better solutions within a reasonable timeout. A specialized branch and bound algorithm for MCM [5] is able to compute solutions for the same MCM benchmark suite in slightly less time than our approach, but it has the downside of not supporting bit-level costs and of not scaling well to larger coefficient word sizes. For the first time, the use of negative fundamentals is exploited within an optimization method and thoroughly evaluated, showing cases with significant bit-level cost reductions. This allowed us to find MCM circuits with less bit-level costs than another ILP-based approach for bit-level cost optimization [12].

Future work could be about leveraging the speed of contemporary SAT solvers to further improve the state-of-the-art of optimal constant multiplication. Our approach could be extended towards (i) constant-matrix-multiplication (CMM) or (ii) optimizing constant multiplication circuits using ternary adders. Regarding bit-level costs, one could (iii) include the input word size into the SAT formulation to allow the solver to remove an adder entirely by choosing a shift larger than the input word size, (iv) extend the SAT formulation to truncated outputs, in cases where the full precision is not necessary, or (v) derive a detailed cost model for ASICs, which must differentiate costs for full adders, half adders and inverters.

References

[1] P. Cappello and K. Steiglitz, "Some complexity issues in digital signal processing," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 32, no. 5, pp. 1037–1041, Oct. 1984.
[2] M. R. Garey, *Computers and intractability : a guide to the theory of NP-completeness*, 27th ed., ser. A series of books in the mathematical sciences. New York, NY: Freeman, 2005.
[3] Y. Voronenko and M. Püschel, "Multiplierless multiple constant multiplication," *ACM Transactions on Algorithms*, vol. 3, no. 2, pp. 11–es, 2007.
[4] M. Kumm, "Optimal Constant Multiplication Using Integer Linear Programming," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 5, pp. 567–571, May 2018.
[5] L. Aksoy, E. O. Güneş, and P. Flores, "Search algorithms for the multiple constant multiplications problem: Exact and approximate," *Microprocessors and Microsystems*, vol. 34, no. 5, pp. 151–162, Aug. 2010.
[6] J. Thong and N. Nicolici, "A novel optimal single constant multiplication algorithm," in *Design Automation Conference*, Jun. 2010, pp. 613–616, iSSN: 0738-100X.

[7] V. Lagoon and A. Metodi, “Deriving Optimal Multiplication-by-Constant Circuits With A SAT-based Constraint Engine,” in *The 19th workshop on Constraint Modelling and Reformulation*, 2020, pp. 1–6.

[8] K. Johansson, O. Gustafsson, and L. Wanhammar, “A detailed complexity model for multiple constant multiplication and an algorithm to minimize the complexity,” in *Proceedings of the 2005 European Conference on Circuit Theory and Design, 2005.*, vol. 3, 2005, pp. III/465–III/468 vol. 3.

[9] ——, “Bit-Level Optimization of Shift-and-Add Based FIR Filters,” in *2007 14th IEEE International Conference on Electronics, Circuits and Systems*, 2007, pp. 713–716.

[10] L. Aksoy, E. Costa, P. Flores, and J. Monteiro, “Optimization of area in digital FIR filters using gate-level metrics,” in *2007 44th ACM/IEEE Design Automation Conference*, ser. Proceedings of the 44th annual conference on Design automation - DAC ’07. Design Automation Conference (DAC), 00 2007, pp. 420–423.

[11] R. Garcia, A. Volkova, and M. Kumm, “Truncated Multiple Constant Multiplication with Minimal Number of Full Adders,” in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2022.

[12] R. Garcia and A. Volkova, “Toward the Multiple Constant Multiplication at Minimal Hardware Cost,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70, no. 5, pp. 1976–1988, May 2023.

[13] X. Lou, Y. J. Yu, and P. K. Meher, “High-Speed Multiplier Block Design Based on Bit-Level Critical Path Optimization,” *IEEE International Symposium of Circuits and Systems (ISCAS)*, pp. 1308 – 1311, 00 2014.

[14] ——, “Fine-Grained Critical Path Analysis and Optimization for Area-Time Efficient Realization of Multiple Constant Multiplications,” *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 62, no. 3, pp. 863 – 872, 07 2015.

[15] M. Kumm, O. Gustafsson, M. Garrido, and P. Zipf, “Optimal Single Constant Multiplication Using Ternary Adders,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 7, pp. 928–932, Jul. 2018.

[16] A. G. Dempster and M. D. Macleod, “Constant integer multiplication using minimum adders,” *IEE Proceedings - Circuits, Devices and Systems*, vol. 141, no. 5, pp. 407–413, Oct. 1994, publisher: IET Digital Library.

[17] O. Gustafsson, “Towards Optimal Multiple Constant Multiplication: A Hypergraph Approach,” in *2008 42nd Asilomar Conference on Signals, Systems and Computers*, ser. Asilomar Conference on Signals, Systems and Computers (ACSSC), 10 2008, pp. 1805 – 1809.

[18] O. Gustafsson, A. G. Dempster, M. D. Johansson, K .and Macleod, and L. Wanhammar, “Simplified design of constant coefficient multipliers,” *Circuits, Systems, and Signal Processing*, vol. 25, no. 2, pp. 225 – 251, 2006.

[19] O. Gustafsson, K. Johansson, and L. DeBrunner, “Techniques for Avoiding Sign-Extension in Multiple Constant Multiplication,” in *2009 Conference Record of the Forty-Third Asilomar Conference on Signals, Systems and Computers*, ser. Asilomar Conference on Signals, Systems and Computers (ACSSC). Asilomar Conference on Signals, Systems and Computers (ACSSC), 00 2009, pp. 740 – 743.

[20] N. Ryzhenko and S. Burns, “Standard cell routing via boolean satisfiability,” in *Proceedings of the 49th Annual Design Automation Conference on - DAC ’12*. San Francisco, California: ACM Press, 2012, p. 603.

[21] T. Iizuka, M. Ikeda, and K. Asada, “High speed layout synthesis for minimum-width CMOS logic cells via Boolean satisfiability,” in *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC ’04. Yokohama, Japan: IEEE Press, Jan. 2004, pp. 149–154.

[22] T. Welp, S. Krishnaswamy, and A. Kuehlmann, “Generalized SAT-sweeping for post-mapping optimization,” in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC ’12. New York, NY, USA: Association for Computing Machinery, Jun. 2012, pp. 814–819.

[23] H. Fraisse, A. Joshi, D. Gaitonde, and A. Kaviani, “Boolean Satisfiability-Based Routing and Its Application to Xilinx UltraScale Clock Network,” in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Monterey California USA: ACM, Feb. 2016, pp. 74–79.

[24] G.-J. Nam, K. A. Sakallah, and R. A. Rutenbar, “Satisfiability-based layout revisited: detailed routing of complex FPGAs via search-based Boolean SAT,” in *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, ser. FPGA ’99. New York, NY, USA: Association for Computing Machinery, Feb. 1999, pp. 167–175.

[25] R. G. Wood and R. A. Rutenbar, “FPGA routing and routability estimation via Boolean satisfiability,” in *Proceedings of the 1997 ACM fifth international symposium on Field-programmable gate arrays*, ser. FPGA ’97. New York, NY, USA: Association for Computing Machinery, Feb. 1997, pp. 119–125.

[26] W. A. Hunt and E. Reeber, “A SAT-based procedure for verifying finite state machines in ACL2,” in *Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, ser. ACL2 ’06. New York, NY, USA: Association for Computing Machinery, Aug. 2006, pp. 127–135.

[27] E. Reeber and J. Sawada, “Combining ACL2 and an automated verification tool to verify a multiplier,” in *Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, ser. ACL2 ’06. New York, NY, USA: Association for Computing Machinery, Aug. 2006, pp. 63–70.

[28] O. Gustafsson, “Lower bounds for constant multiplication problems,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 54, no. 11, pp. 974 – 978, 11 2007.

[29] N. Eén and N. Sörensson, “Translating Pseudo-Boolean Constraints into SAT,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, no. 1-4, pp. 1–26, Mar. 2006.

[30] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, “CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020,” in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.

[31] Gurobi, “Gurobi Optimizer,” 2022.

[32] L. de Moura and N. Bjørner, “Z3: An Efficient SMT Solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer, 2008, pp. 337–340.

[33] S. Laurent, “Glucose SAT Solver,” 2023.

[34] D. Leothaud, “Hardware acceleration MCM,” https://gitlab.inria.fr/dleothau/hardware-acceleration-mcm, 2022, commit: 7d5063816106ebdd8dda0d33500dcde3237e4c9e.

Nicolai Fiege received the B.Sc. and M.Sc. degrees in Electrical Engineering from the University of Kassel, Germany, in 2018 and 2021, respectively, where he is currently working toward his Ph.D. in Electrical Engineering.

His research interests include the design of optimal digital circuits using mathematical methods and frameworks.

Martin Kumm received the Dipl.-Ing. degree in electrical engineering from the Technical University of Darmstadt, Germany in 2007.

He was with GSI Darmstadt, working on digital control systems for particle accelerators from 2003 to 2009. In 2015, he received his Ph.D. (Dr.-Ing.) degree from the University of Kassel, Germany. He is currently a Professor for Embedded Systems at the Fulda University of Applied Sciences, Germany. His research interest is application-specific arithmetic and its optimization with particular emphasis on reconfigurable systems.

Peter Zipf (M’05) received the Ph.D. (Dr.-Ing.) degree from the University of Siegen, Germany, in 2002.

He was a Postdoctoral Researcher at the Department of Electrical Engineering and Information Technology, Darmstadt University of Technology, Darmstadt, Germany, until 2009. He is currently the chair of Digital Technology at the University of Kassel, Germany. His current research interests include reconfigurable computing, embedded systems and CAD algorithms for circuit optimization.