Recurrence Relations Revisited: Scalable Verification of Bit Level Multiplier Circuits

Amr Sayed-Ahmed[*], Ulrich Kühne[*], Daniel Große[*], and Rolf Drechsler[*][†]
[*] Faculty of Mathematics and Computer Science, University of Bremen, Germany
[†] Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
Email: {asahmed, ulrichk, grosse, drechsle}@informatik.uni-bremen.de

Abstract—**Although a lot of effort has been spent on verifying arithmetic designs, it is still a problem that has no general robust automated solution. One major challenge is verifying large scale multiplier circuits. For this purpose, we revisit the idea of using functional properties of the multiplication function, which can be expressed by recurrence equations. Then, instead of proving the equivalence of the implementation and a specification, the verification task is to show that the implementation satisfies the recurrence equation. We propose an approach which makes this verification task practically feasible for large scale multiplier circuits. Based on a combined add/multiply recurrence equation we can make efficient use of case splitting wrt. the partial products of the multiplier. As a result, the problem is split such that only a small part of the multiplier will be checked in every case, thereby avoiding redundant checks between the cases. Overall, our approach allows to verify a variety of multiplier designs in practical time. We present results for multipliers up to 128 bits.**

I. Introduction

Verification of arithmetic circuits is still a big challenge. Since the famous floating point bug in Intel's Pentium processor [1], a lot of effort has been spent on the verification of arithmetic designs. The applied techniques can be divided into simulation-based verification and formal verification techniques. Simulation-based verification techniques can be used to discover bugs in arithmetic designs and they scale very well, but they cannot prove the absence of bugs. For the latter, formal methods are needed [2]. While large arithmetic circuits – in particular floating-point logic – have been verified using interactive theorem proving [3], this requires a huge amount of expert manual work. In this paper we concentrate on automated methods.

Automated equivalence checking – also used in our work – compares a gate level implementation against a reference implementation by combining them into one circuit, called a miter. Most equivalence checking tools exploit structural similarities between the two implementations and try to prove equivalence by decomposing the problem according to the internal equivalences (cut points) [4]. Therefore, equivalence checkers usually cannot deal with implementations that have few internal equivalences. This problem occurs especially for arithmetic circuits, where one function can be implemented in many different ways [5].

Formal verification of large multiplier designs is still a hard problem that has no general automated solution. Without any information on the high level structure of the netlists, property checking and many equivalence checking approaches fail to verify these circuits. Most successful techniques make use of knowledge about the architecture of the multiplier (see e.g. [6]). In this paper, we address the problem of verifying large multipliers on the gate level without any information about their high level structure. For this problem, several approaches have been proposed in the past. However, most of these approaches have a scalability problem, cannot be applied on all types of multiplier architectures, or they fail if the circuit does not represent a correct multiplier function. The existing approaches can be divided mainly into three categories: 1) decision diagrams, 2) reverse engineering and structural methods, and 3) approaches based on special arithmetic properties of the realized function.

Decision diagrams provide a canonical representation for netlists of the *design under verification* (DUV) and the *reference implementation* (RI). The equivalence check then consists of checking that the resulting decision diagrams are identical. Unfortunately, many popular data structures like *Reduced Ordered Binary Decision Diagrams* (ROBDDs) [7] have exponential size for multiplication. *Multiplicative Binary Moment Diagrams* (*BMDs) [8] can represent the word level multiplier function in a compact way. However, exponential blowup can still occur for incorrect designs or during the construction of the *BMD from a bit-level circuit [9].

The second direction in the classification is based on reverse engineering. An approach in this direction has been proposed in [5]. It extracts adder structures from bit level netlists and builds half adder networks. This approach has mainly two drawbacks: First, it makes assumptions on the internal structure of the circuits that are not fulfilled by all multiplier architectures [10]. Second, the approach fails to build the half adder networks if the circuit does not represent a correct multiplier, leading to an inconclusive result. Nevertheless, this technique achieves promising results, e.g. it allows to verify a 48 bit multiplier in about 30 minutes. Other works in this direction use polynomial representations to verify arithmetic circuits [11], [12]. The verification problem is formulated as a reduction of polynomial equations using Gröbner basis. This technique has been used before in verification of Galois field multipliers [13]. The approach can verify large multipliers, however it suffers from the explosion of the number of polynomial terms, when verifying highly optimized circuits [11].

The third direction exploits arithmetic properties of the multiplier function to build a miter with many internal equivalences. The first approach in this context has been proposed by Fujita [14]. It is based on the fact that any function satisfying the recurrence relation $(X + 1) * Y = X * Y + Y$ is a multiplication. However, the original approach by Fujita does not scale, and it cannot verify multipliers larger than 16 bits (see also experiments later). A related approach [15] is based on case splitting by forcing a bit of one of multiplier operands to be zero and summing partial products that belong to this bit outside the multiplier. The problem of this idea is that the similarity of the nets inside the miter structure depends on the order of the partial products of the DUV, therefore this approach only works for certain implementations of multipliers.

The technique presented in this paper belongs to the third category. The proposed approach decomposes the verification of a multiplier such that in every case, the generation and the addition of one partial product of the DUV will be checked using a recurrence equation. In every case, the small difference between the compared implementations and the similar ways of carry propagation allow fast equivalence checking, regardless of the multiplier size. As the multiplier size increases, the number of cases will increase, while the complexity to check one case will remain almost the same. Our approach is able to verify a multiplier at the gate level without any information about its high level specification or the internal structure of the netlist. In addition, it is a general technique that can be applied on various different architectures of the DUV. Overall, it allows to verify large scale multipliers in practical time. The experiments show the ability of the proposed approach to verify 128 bits multiplier.

The remainder of the paper is structured as follows. In Section II, we explain the theoretical background of equivalence checking based on recurrence relations and review the original approach of Fujita as an example on this equivalence checking type. Section III discusses the details of our partial product approach. Experimental results are presented in Sections IV, and we conclude our work in Section V.

II. Equivalence Checking Based on Recurrence Relations

In classical equivalence checking, the DUV and the *reference implementation* (RI) are combined in one circuit called miter by XOR-ing every output pin of the DUV with the respective output pin of the RI, and computing the OR of these functions. An efficient miter should have many similar nets, so the equivalence check can be partitioned into less complex sub-problems.

In this section, we review an equivalence checking that builds the RI from the DUV itself. Therefore it does not need a golden reference model and does not require any knowledge about the internal architecture of the DUV. This technique can be applied on any function which can be defined by a recurrence relation. These functions are also known as *primitive recursive (p.r.) functions* [16]. In the following, we briefly discuss p.r. functions, before showing how to exploit the recurrence relations for equivalence checking. Finally, we give an overview on Fujita's approach [14].

Consider the following theorem, as stated in [16]:

Theorem 1. Given $g \in \mathbb{N}^k \to \mathbb{N}$ and $h \in \mathbb{N}^{k+2} \to \mathbb{N}$, there is a **unique** $f \in \mathbb{N}^{k+1} \to \mathbb{N}$ satisfying:

$$f(0, \vec{B}) = g(\vec{B})$$

$$f(A+1, \vec{B}) = h(f(A, \vec{B}), A, \vec{B})$$

for all $\vec{B} \in \mathbb{N}^k$ and $A \in \mathbb{N}$.

Based on this theorem, a unique p.r. function f is defined, if it can be constructed from other p.r. functions g and h by composition and primitive recursion. Let $\vec{B} = (B_0, B_1, \cdots, B_{k-1})$ and $A, B \in \mathbb{N}$. The basic p.r. functions are, a) the constant zero $C(\vec{B}) = 0$, b) the projection function $P_i(\vec{B}) = B_i$, and c) the successor function $S(A) = A + 1$. Known examples for p.r. functions are addition and multiplication. The addition function $Add(A, B)$ is defined according to Theorem 1 by the relations $Add(0, B) = B$ and $Add(A + 1, B) = S(Add(A, B))$. Note that the first relation is given in terms of a projection function, while the second one uses the successor function, which are both basic p.r. functions.

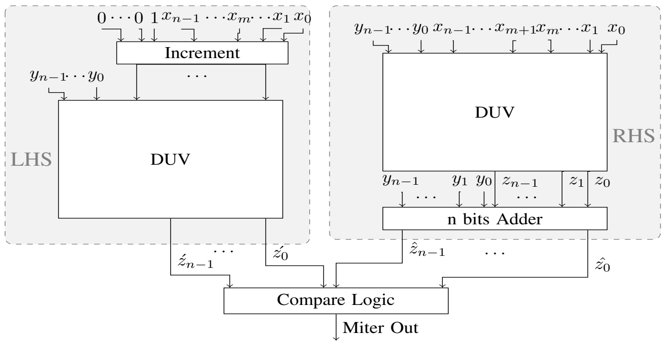
0 ⋯ 0 1 $x_{n-1} \cdots x_m \cdots x_1$ x_0
Increment
$y_{n-1} \cdots y_0$
⋯
LHS
DUV
$\acute{z}_{n-1}$
$\acute{z}_0$
$y_{n-1} \cdots y_0$ $x_{n-1} \cdots x_{m+1} x_m \cdots x_1$ x_0
DUV
RHS
y_{n-1} y_1 y_0 z_{n-1} z_1 z_0
n bits Adder
$\hat{z}_{n-1}$
$\hat{z}_0$
Compare Logic
Miter Out

Fig. 1: The Equivalence Checking Miter of a Multiplier using Fujita's approach

This implies that Add is a p.r. function as well. Based on this, the multiplication function $Mult(A, B) = A * B$ is uniquely defined by $Mult(0, B) = 0$ and $Mult(A + 1, B) = Add(Mult(A, B), B)$, and is therefore also a p.r. function.

The uniqueness of a p.r. function thus defined gives rise to an equivalence checking method. Given some implementation F of a p.r. function f, we can check that F correctly realizes f by checking that it satisfies the recurrence relations of f. In case of the multiplication function, if we succeed to show that F satisfies the relations $F(0, B) = 0$ and $F(A+1, B) = Add(F(A, B), B)$ for any inputs A and B, this implies that F is indeed a multiplication. For most p.r. functions, checking the first relation is relatively easy, while the second relation is verified by building a miter according to the recurrence relation and using standard equivalence checking techniques to show that both sides of the equation are equal for arbitrary inputs. In the following, we will refer to this approach as *equivalence checking based on recurrence relations* (ECRC).

An example of ECRC has been proposed by Fujita in [14]. The approach verifies a multiplier circuit by checking the equivalence of both sides of the recurrence relation:

$$Mult(X + 1, Y) = Mult(X, Y) + Y.$$

The resulting miter structure is shown in Figure 1. An implementation of the *left hand side* (LHS) of the equation is built by adding an increment circuit to the first operand X of the DUV, while the *right hand side* (RHS) is implemented by adding the second operand Y to to the output Z. The equivalence checking compares the two circuits in order to prove that the DUV is indeed a multiplier.

Although the two sides use the same circuit for multiplication, they have different inputs: $(X + 1, Y)$ on the LHS and (X, Y) on the RHS. Therefore the equivalence checker will not be able to find enough internal equivalence points. Fujita overcomes this problem by splitting the checking process into a series of sub-problems. The case split is performed wrt. to the position at which the last carry occurs on the output of the increment circuit $(X + 1)$. It can be observed that in the bit vector representing $X + 1$, the value of an input bit x_m will be inverted iff all lower bits $x_i, 0 \le i \le m - 1$ are equal to 1. Thus, for each case the increment circuit can be replaced by simple inverters for the $m + 1$ lower bits of X. The higher bits $x_i, i > m$ will not be modified. However, for larger multipliers beyond 16 bit, this simplification of the increment circuit does not lead to sufficient similarities to enable a scalable verification.

III. Checking Partial Product Approach

In this section, we present our approach to verify an implementation of an integer multiplier. After some basic definitions of the multiplication function and the combined add/multiply function, we give an overview of the proposed approach. Then, we present the theoretical part of the approach. Finally, the implementation of the proposed approach will be introduced.

A. Basic Notions

We denote an integer multiplier as $Z = X*Y$, X and Y are the integer operands of the multiplier, Z is the integer result of the multiplier. These integer variables will be represented as vectors of Boolean variables, such that $X = \sum_{i=0}^{n-1} 2^i x_i$, $Y = \sum_{i=0}^{n-1} 2^i y_i$, and $Z = \sum_{i=0}^{2n-1} 2^i z_i$, where n is number of Boolean bits that represent each operand of the multiplier, 2^i is the weight of each Boolean variable. The multiplier $Z = X*Y$ can be expressed on the bit level as

$$\sum_{i=0}^{2n-1} 2^i z_i = \sum_{i=0}^{n-1} 2^i x_i * \sum_{i=0}^{n-1} 2^i y_i.$$

Typically, the multiplication of two operands is performed by generating *partial products*, which are then summed using an addition tree as shown in Figure 2.

Definition 1. A *partial product* is a bitwise multiplication $pp_{ci,ri} = y_{ci-ri} * x_{ri}$, where ri and ci are the row/column indices of the partial product in the addition tree. The partial product $pp_{ci,ri}$ has weight 2^{ci}, which is the product of the weights of x_{ri} and y_{ci-ri}. For input bit width n, we define that

$$pp_{ci,ri} = \begin{cases} y_{ci-ri} * x_{ri} & 0 \leq ci - ri \leq n-1 \\ 0 & Otherwise \end{cases}$$

As an example, Figure 2 shows the partial product $pp_{4,4} = y_0 * x_4$.

The *partial products generator* generates partial products, such that every bit of the first operand is multiplied with all bits of the second operand. The generated partial products satisfy

$$\sum_{ci=0}^{2n-2} (2^{ci} * \sum_{ri=0}^{n-1} pp_{ci,ri}) = \sum_{i=0}^{n-1} 2^i x_i * \sum_{i=0}^{n-1} 2^i y_i.$$

The *addition tree* is the addition of partial products to generate the multiplier result Z. The addition is done according to the weights of the partial products, such that

$$\sum_{i=0}^{2n-1} 2^i z_i = \sum_{ci=0}^{2n-2} (2^{ci} * \sum_{ri=0}^{n-1} pp_{ci,ri}).$$

Note that $pp_{ci,ri}$ is equal to zero, if $ci - ri < 0$ or $ci - ri > n-1$. The tree has two indices ri and ci, where ri is the index of the tree rows, while ci is the index of the tree columns. Partial products are ordered in the tree as shown in Figure 2, such that partial products have the same weight 2^{ci} if they belong to the same column ci. The output bit z_{ci} belongs to the column ci, having also weight 2^{ci}.

The *addition tree equation* is the equation that formulates the addition of the elements of every column in the tree and the cone that generates the output bit of a column. The addition of partial products in every column generates an output bit as a sum and carry bits. These carry bits are propagated to the next column. Therefore a column does not only take into account the partial products that belong to this column, but also carry bits propagated from the previous column. The addition of partial products in every column ci to the propagated carry bits from the previous column $ci-1$, generates the output bit z_{ci} as sum and new carry bits to the next column $ci+1$. We

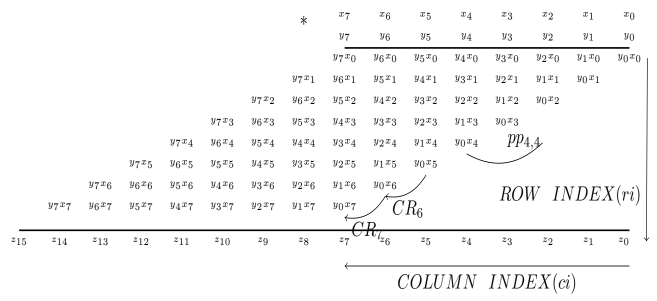

Fig. 2: Addition tree of a 8 bit multiplier

denote the summation of these carry bits as CR_{ci+1}, and the summation of the carry bits that propagate to the column ci as CR_{ci}. Based on that, we formulate the addition tree equation of column ci as

$$z_{ci} + 2 * CR_{ci+1} = CR_{ci} + \sum_{ri=0}^{n-1} pp_{ci,ri}. \quad (1)$$

As an example consider again Figure 2, where CR_6 is the summation of carry bits which propagate to column 6, and CR_7 is the summation of carry bits which are generated from column 6 and propagate to column 7.

In Section II we have explained how we can verify primitive recursive functions using the ECRC technique. In our approach, we use a p.r. function to verify the generation and addition of partial products. Verifying a multiplier then boils down to check the whole addition tree based on this method. For this purpose, we define the *combined multiply-add* (CMA) function, and show that CMA is a p.r. function, therefore its implementation can be verified using its recurrence relations.

Definition 2. The *combined multiply-add* (CMA) function $CMA(a_0, a_1, B) = a_0 * a_1 + B$, combines the bitwise multiplication function and the addition function, where B is an integer quantity, a_0 and a_1 are Boolean variables.

Lemma 1. The CMA function $CMA(a_0, a_1, B)$ is a p.r. function. **Proof**: According to Theorem 1, the CMA function can be defined by the relations $CMA(0, a_1, B) = B$ and $CMA(a_0 + 1, a_1, B) = Add(CMA(a_0, a_1, B), a_1)$. The first relation is a projection, while the addition function Add of the second relation is known to be a p.r. function. This implies that the CMA function is itself a p.r. function.

B. Overview of the Approach

As shown in the previous section a multiplier can be implemented as an addition tree of partial products, which are generated using bitwise multiplications. Based on that, the proposed approach applies case splitting by checking in each case the correct generation of only one partial product and the correct addition of this partial product to other partial products of the multiplier. We will refer to this partial product as *Partial Product Under Verification* (PPV). The combination of the generation and the addition of PPV can be defined using the CMA function from Definition 2. As the CMA function is a p.r. function, its implementation can be checked using ECRC.

The approach applies ECRC on the cone that is influenced by generating and adding PPV to other partial products. We refer to this cone as the PPV cone. This PPV cone can be described using the addition tree and the addition tree equation, as defined in Subsection III-A. As stated there, every partial product belongs to a column in the addition tree, and this column is formulated using the addition tree equation.

The column containing the PPV receives carry bits from the previous column. Adding PPV to these carry bits and to other partial products of the column will generate a sum bit and carry bits, which propagate to the next column. Based on that the PPV cone of generating and adding PPV will be the gates that are represented by, a) the columns that produce carry bits to the column of PPV, b) the column that receives carry bits from the column of PPV, and c) the column of PPV itself.

The approach decomposes the verification into n^2 cases, which corresponds to the number of partial products in an n bit multiplier. In every case, it extracts the PPV cone and applies equivalence checking based on the CMA recurrence relation. This case split leads to small differences between the compared implementations of the constructed miter in each case and the approach avoids redundant checks between the cases, which allows for fast equivalence checking regardless of the multiplier size. In the following, we will refer to this approach as the *Checking Partial Product* (CPP) approach.

C. Mathematical Formulations

The goal of the approach in each case split is to verify the combination of generating and adding the PPV. The generation and the addition of PPV is a CMA function $a_0 * a_1 + B$. The approach verifies this function by extracting its implementation from the DUV (the PPV cone), and checking the consistency of the cone with the CMA relations. Consider Definition 1 and Eq. (1) to formulate a mathematical expression for the PPV cone. PPV is a partial product, therefore it can be expressed as $pp_{cv,rv} = x_{rv} * y_{cv-rv}$, where cv and rv are indices of PPV. cv and rv are ranging over the ranges of the column index ci and the row index ri. PPV belongs to a column in the addition tree which has been formulated using the addition tree equation. The addition tree equation of PPV after the extraction of PPV outside the summation of other partial products will be

$$z_{cv} + 2 * CR_{cv+1} = CR_{cv} + \sum_{\substack{ri=0 \\ ri \neq rv}}^{n-1} pp_{cv,ri} + pp_{cv,rv} \quad (2)$$

As can be seen from Eq. (2), adding PPV generates the carry CR_{cv+1} that propagates to column $cv + 1$. Therefore PPV addition has influence on column $cv + 1$. The addition tree equation of this column is

$$z_{cv+1} + 2 * CR_{cv+2} = CR_{cv+1} + \sum_{ri=0}^{n-1} pp_{cv+1,ri} \quad (3)$$

Because the structural relations between the addition tree equations are summations, the formulation of the PPV cone will be the summation of Eq. (2) that formulates the cone of z_{cv}, to Eq. (3) that formulates the cone of z_{cv}. Note that column $cv + 1$ has higher weight with 2^1 than column cv, therefore Eq. (3) will be multiplied by 2, then it is summed to Eq. (2), which leads to the PPV cone equation

$$z_{cv} + 2 * z_{cv+1} + 4 * CR_{cv+2} = \\ CR_{cv} + \sum_{\substack{ri=0 \\ ri \neq rv}}^{n-1} pp_{cv,ri} + \sum_{ri=0}^{n-1} 2 * pp_{cv+1,ri} + pp_{cv,rv} \quad (4)$$

Note that the term $2 * CR_{cv+1}$ is removed from the two sides of Eq. (4). We refer to the integer quantity that are added to PPV as

$$Q_{rv} = CR_{cv} + \sum_{\substack{ri=0 \\ ri \neq rv}}^{n-1} pp_{cv,ri} + \sum_{ri=0}^{n-1} 2 * pp_{cv+1,ri},$$

and we substitute the PPV term $pp_{cv,rv}$ by $x_{rv} * y_{cv-rv}$, which reformulates Eq. (4) to

$$z_{cv} + 2 * z_{cv+1} + 4 * CR_{cv+2} = x_{rv} * y_{cv-rv} + Q_{rv} \quad (5)$$

The right side of Eq. (5) formulates the mathematical expression of the PPV cone. Note that the PPV cone is a bitwise multiplication followed by addition, therefore it is an implementation of the CMA function. By replacing the term a_0 of the CMA recurrence relation with x_{rv}, the term a_1 with y_{cv-rv}, and the term B with Q_{rv}, we apply the CMA relations on the PPV cone, such that Eq. (5) becomes

$$z_{cv} + 2 * z_{cv+1} + 4 * CR_{cv+2} = CMA(x_{rv}, y_{cv-rv}, Q_{rv})$$

The initial relation of PPV will be $CMA(0, y_{cv-rv}, Q_{rv}) = Q_{rv}$, by assigning zero to x_{rv}. As the value of Q_{rv} depends on the value of other partial products of the multiplier, checking the initial relation for every partial product separately is not trivial. The approach overcomes this problem by checking the initial relations of all partial products together. Because in every initial relation $x_{rv} = 0$, checking all initial relations together, is done by assigning zeros to all x_i bits, which implies that $X = \sum_{i=0}^{n-1} 2^i x_i = 0$. At $X = 0$, the approach checks trivially that the result of the DUV is equal to zero, where $0 * Y = 0$. Therefore checking the initial CMA relations together is done, by checking the initial relation of the multiplier.

After checking the initial relation of PPV, the recurrence relation of PPV is checked which is

$$CMA(x_{rv} + 1, y_{cv-rv}, Q_{rv}) = \\ CMA(x_{rv}, y_{cv-rv}, Q_{rv}) + y_{cv-rv} \quad (6)$$

The approach checks this recurrence relation for every PPV cone, by applying equivalence checking. The approach builds a miter, such that the implementation of the miter represents the left side of the recurrence relation of PPV, while the other implementation represents the right side of this relation. The implementation details of the miter will be explained in the next subsection.

By checking the addition and the generation of all partial products in the multiplier, the approach announces the consistency of the DUV with the multiplication function.

D. Implementation

Implementing the recurrence relation of Eq. (6) as it is, in every splitting case, will suffer from redundant checks. The term $x_{rv} + 1$ propagates a carry bit of value one to other bits of X which has higher weights, like $x_{rv+1}, x_{rv+2}, \cdots, x_{n-1}$. As the value one will be added to these bits in other splitting cases, it is redundant to implement the term $x_{rv} + 1$ as an addition. To simplify the implementation of the equivalence checking and remove these redundant checks, the approach assigns zero to the bit x_{rv}, at this case $x_{rv} + 1$ does not generate a carry bit. Therefore the one bit adder $x_{rv} + 1$ in the left side of Eq. (6) is implemented by the XOR function $x_{rv} \oplus 1$, which is the inversion of x_{rv} ($\bar{x}_{rv}$). Because of this optimization the carry bit that results from $x_{rv} + 1$ to higher bits will be stopped, and building the equivalence checking miter will be easier. At $x_{rv} = 0$, still all patterns of the $pp_{cv,rv}$ are checked, where $pp_{cv,rv} = 1 * y_{cv-rv}$ in the implementation of of the left side of Eq. (6), and $pp_{cv,rv} = 0 * y_{cv-rv}$ in the implementation of the right side of Eq. (6), which implies that this assigning will not affect the coverage of the approach.

We implemented our approach using the tool ABC presented in [17]. The approach applies n^2 case splits. In every case, it builds a miter circuit based on Eq. (6), which checks

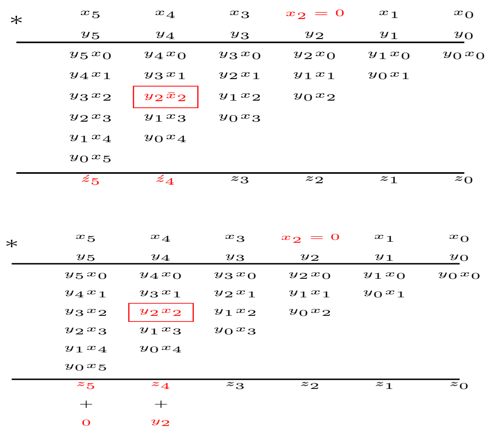

Fig. 3: The miter inputs for checking $pp_{4,2} = y_2 * x_2$

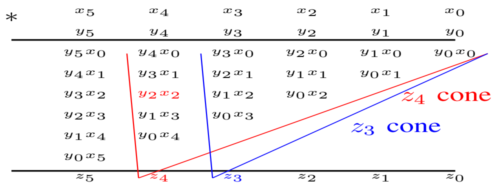

Fig. 4: The search of CPP approach for $pp_{4,2} = y_2 * x_2$

the generation and the addition of one partial product PPV, by applying the following steps:

1) It extracts two copies of the cone of $z_{cv} + 2 * z_{cv+1}$, which are the gates that are connected to the output bits z_{cv} and z_{cv+1}. These gates represent the PPV cone.
2) It assigns zero to the bit x_{rv} of the two copies, which is x_2 in the example of Figure 3 (upper and lower part).
3) In the first copy, it searches for the partial product $pp_{cv,rv}$ and inverts explicitly the bit x_{rv} of this partial product. This is shown in the upper part of Figure 3, where it inverts the bit x_2 of the partial product $x_2 * y_2$.
4) In the second copy, it adds to output bits z_{cv} and z_{cv+1}, the input bit y_{cv-rv}, using a 2 bit adder. This can be seen in the lower part of Figure 3, where the input bit y_2 is added to output bits $z_4 + 2 * z_5$.
5) The miter is completed with the usual comparison logic by XOR-ing and OR-ing the two output bits of the first copy with two output bits of the adder that is added to the second copy.

Checking the two implementations which are based on the recurrence relation of Eq. (6) is very fast because a) the carry CR_{cv} in both implementations propagates in the same way, b) the carry CR_{cv+1} which has different propagation ways in the compared implementations does not propagate through all the multiplier implementations, c) the difference between compared implementations is in the generation of PPV and the propagation way of CR_{cv+1}. Note that CR_{cv+1} propagates in the first copy through the multiplier implementation, while in the second copy it propagates such that parts from it propagate through the multiplier implementation and the other part propagates through the added Adder. Equivalence checking techniques like *rewriting* and *fraiging* [18] succeed to prove the equivalence (or non-equivalence) of many cases without resorting to the SAT solver. This explains the effectiveness of the CPP approach, even when applied to multipliers larger than 32 bits.

By our case splitting scheme, a total of n^2 miters are constructed. These miters can be checked in parallel or serially, since the cases are independent of each other. We use parallelism for multipliers larger than 32 bits. In general, the CPP approach shows good results in the verification of multipliers of various architectures, as is discussed in the next section.

However, the approach suffers from some limitations. It cannot verify Booth recoding multipliers and some optimized multipliers. To clarify the approach limitations, consider Step 3 of the approach steps. The approach searches for the gate of the partial product $pp_{cv,rv}$. The search is done by comparing gates of cone z_{cv} and cone z_{cv-1}. The gate that belongs to the cone of z_{cv}, does not belong to the cone z_{cv-1}, and has the input bit x_{rv}, is the one that the approach searches for. Figure 4 shows an example of this search. After finding the gate of the PPV, the approach inverts the input bit x_{rv} of this gate. This inversion does not always affect on the partial product $pp_{cv,rv}$, but may affect other partial products, which is the case for multipliers based on Booth recoding and some optimized multipliers.

IV. Experimental Results

Our approach is built on top of the ABC tool [18]. ABC compiles the miter circuit into an *And-Inverter Graph* (AIG) and applies structural reduction techniques like *fraiging* and *rewriting*. These techniques reduce effectively the size of the AIG if the miter has many similar internal nodes. At the backend of the equivalence checking procedure, the reduced AIG is converted to *Conjunctive Normal Form* (CNF) and the resulting instance is given to MiniSAT [19].

All experiments have been carried out on an Intel(R) Core(TM) i5-3320M CPU (2.6 GHz, 16 GByte) running Linux. For the experiments, we generated different multiplier architectures using the online tool *Arithmetic Module Generator* [20]. The multipliers are given as Verilog RTL code. The synthesis of the designs to a gate level netlists has been done using the *Yosys Open Synthesis Suite* [21] and ABC.

The multiplier architectures are categorized according to 1) the type of the partial products generator, 2) the partial products accumulator, and 3) the last stage adder. In our experiments, we use one type of partial products generator, namely *simple partial products* (denoted by SPP). The types of partial products accumulator are *array* (ARR), *wallace tree* (WLT), and (4,2) *compressor tree* (42CT). Finally, the chosen types of the last stage adder are *ripple carry adder* (RCA), *carry look-ahead adder* (CLA), and *brent-kung adder* (BKA). In the following the benchmarks are named according to their architecture features. For example, a circuit with simple partial products, a wallace tree as partial products accumulator, and a ripple carry adder as last stage adder will be labeled as SPP-WLT-RCA.

We conducted two types of experiments. The first one to verify the practical time of the approach in checking larger multipliers. The second one to show the capability of discovering bugs.

A. Equivalence Checking Results

We compare the run-times of our CPP approach against Fujita's approach. In the original approach of Fujita the equivalence is checked using ROBDDs. Here, – and for a fair comparison – we use Fujita's approach with ABC as backend for equivalence checking.

The first column of Table I shows the name of the circuit. The second column gives the number of inputs and output bits. The next two columns provide the run-times. Note that the run-times of Fujita's approach include only the verification

TABLE I: Run times for verification of multipliers

Benchmark	I/O bits	CPP [h:m:s]	Fujita [h:m:s]
SPP-ARR-RCA	16/32	00:01:23	00:00:31
SPP-WLT-CLA	16/32	00:00:46	00:09:08
SPP-WLT-BKA	16/32	00:00:52	00:10:05
SPP-42CT-CLA	16/32	00:00:44	00:49:37
SPP-42CT-BKA	16/32	00:00:43	00:17:56
SPP-ARR-RCA	32/64	02:34:40	11:09:18
SPP-WLT-CLA	32/64	00:15:12	TO
SPP-42CT-BKA	32/64	00:21:20	TO
SPP-ARR-RCA	48/96	20:32:12	TO
SPP-WLT-CLA	48/96	01:29:36	TO
SPP-42CT-BKA	48/96	01:20:00	TO
SPP-ARR-RCA	64/128	94:37:20	TO
SPP-WLT-CLA	64/128	05:46:40	TO
SPP-42CT-BKA	64/128	05:31:44	TO
SPP-42CT-BKA	128/255	78:11:12	TO

TABLE II: Results for models with injected faults

Benchmark	I/O bits	|AIG|	#Faults	∅ runtime
SPP-ARR-RCA	16/32	2126	4252	$2.56\,s$
SPP-WLT-CLA	16/32	2988	5976	$1.80\,s$
SPP-42CT-BKA	16/32	2201	4402	$0.45\,s$
SPP-WLT-CLA	32/64	14196	710	$27.05\,s$
SPP-WLT-CLA	32/64	12741	319	$59.72\,s$
SPP-42CT-BKA	32/64	9173	459	$10.00\,s$

of the lower n bits (not the full $2n$ output bits). The time out (TO in the table) has been set to 100 hours. Please note that for a naive miter construction (one big miter) and then running the ABC command (CEC) *all* benchmarks timed out after 100 hours.

The experiments show that the verification time of the CPP approach depends not only on the size of the multiplier circuit, but also on the type of the partial products accumulator. The circuits with wallace tree or (4,2) compressor are verified in less time than those with array accumulator. As can be seen our approach verifies the correctness of the multipliers for up to 32 bits in practical time. Fujita's approach fails here already for the complex architectures. Furthermore, our approach allows the verification of multipliers up to 128 bits.

B. Fault Injection

In order to demonstrate the ability to discover bugs, we applied our approach to faulty designs that have been created by automatic fault injection. The faults have been injected in the netlist which is given as an AIG. We applied the CPP approach on different copies of each netlist where each copy contains one single fault. The approach has succeeded to discover all bugs. The results are summarized in Table II. The first two columns describes the type of the multiplier architecture (as explained in the previous section) and the bit width, respectively. The third column gives the size (number of nodes) of the AIG. The number of performed runs is given in the fourth column, and the average run-time needed to discover the bug is given in the last column.

For all 16 bit architectures, we systematically covered the whole AIG by injecting two faults for each node. For the larger designs, random gates were chosen with an even distribution over the netlist. For the 32 bit multipliers it can be observed that the run-times vary between fractions of a second and several minutes, where usually more than half of the bugs are discovered in less than a second.

To summarize, the results show that our approach works well for bug-hunting as well as for the verification of correct multiplier designs.

V. Conclusion

Verification of bit level multipliers still has no general automated solution. In this paper, we verify multipliers at the gate level without any information about their high level designs. We have developed an approach that allows to verify multiplier circuits up to 128 bits. The approach is based on functional properties of the multiplication function, which can be expressed as recurrence equation, as well as a new case splitting scheme. As a consequence enough similarities remain for the equivalence check of each case. Overall, the approach increases the scalability of equivalence checking to verify larger multipliers, however it cannot be applied for all types of multiplier architectures.

Acknowledgment

This work was supported in part by the German Research Foundation (DFG) (DR 287/23-1) within a Reinhart Koselleck project, by the University of Bremens graduate school SyDe, funded by the German Excellence Initiative and by the German Federal Ministry of Education and Research (BMBF) within the project Effektiv under contract no. 01IS13022E.

References

[1] H. P. Sharangpani and M. L. Barton, "Statistical analysis of floating point flaw in the pentium processor(1994)," Intel, Tech. Rep., Nov. 1994.

[2] A. Sayed-Ahmed, H. Fahmy, and U. Kühne, "Verification of the decimal floating-point square root operation," in *ETS*, 2014, pp. 1–2.

[3] J. Harrison, "Floating-point verification," *Journal of Universal Computer Science*, vol. 13, pp. 629–638, 2007.

[4] A. Kühlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *DAC*, 1997, pp. 263–268.

[5] D. Stoffel and W. Kunz, "Equivalence checking of arithmetic circuits on the arithmetic bit level," *IEEE Trans. on CAD*, pp. 586–597, 2004.

[6] B. Xue, P. Chatterjee, and S. K. Shukla, "Simplification of c-rtl equivalent checking for fused multiply add unit using intermediate models," in *ASPDAC*, 2013, pp. 723–728.

[7] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys 24(3)*, pp. 293–318, 1992.

[8] R. E. Bryant and Y. A. Chen, "Verification of arithmetic circuits with binary moment diagrams," in *DAC*, 1995.

[9] K. Hamaguchi, A. Morita, and S. Yajima, "Efficient construction of binary moment diagrams for verifying arithmetic circuits," in *ICCAD*, 1995, pp. 78–82.

[10] D. Stoffel, E. Karibaev, I. kufareva, and W. Kunz, *Advanced Formal Verification*, R. Drechsler, Ed. Kluwer Academic Publishers, 2004.

[11] M. Ciesielski, C. Yu, D. Liu, and W. Brown, "Verification of gate-level arithmetic circuits by function extraction," in *DAC*, 2015.

[12] F. Farahmandi and B. Alizadeh, "Gröbner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction," *MICPRO Journal*, vol. 39, no. 2, pp. 83–96, 2015.

[13] J. Lv, P. Kalla, and F. Enescu, "Efficient gröbner basis reductions for formal verification of galois field multipliers," in *DATE*, 2012, pp. 899–904.

[14] M. Fujita, "Verification of arithmetic circuits by comparing two similar circuits," in *CAV*, ser. LNCS, vol. 1102. Springer Verlag, 1996, pp. 159–168.

[15] Y.-T. Chang and K.-T. Cheng, "Self-referential verification of gate-level implementations of arithmetic circuits," in *DAC*, 2002, pp. 311–316.

[16] N. Cutland, *Computability: An introduction to recursive function theory*. Cambridge university press, 1980.

[17] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking," in *ICCAD*, 2006, pp. 836–843.

[18] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *CAV*, 2010, pp. 24–40.

[19] N. Een and N. Sörensson, "An extensible sat solver," in *SAT*, ser. LNCS, vol. 2919. Springer Verlag, 2004, pp. 502–518.

[20] "Arithmetic Module Generator Based on ACG," 2014, available at http://www.aoki.ecei.tohoku.ac.jp/arith/.

[21] C. Wolf, "Yosys Open Synthesis Suite," 2014, available at http://www.clifford.at/yosys/.