# Equivalence Checking of Arithmetic Circuits on the Arithmetic Bit Level

Dominik Stoffel, *Member, IEEE,* and Wolfgang Kunz, *Member, IEEE*

***Abstract*—One of the most severe shortcomings of currently available equivalence checkers is their inability to verify arithmetic circuits and multipliers, in particular. In this paper, we present a bit-level reverse-engineering technique that complements standard equivalence checking frameworks. We propose a Boolean mapping algorithm that extracts a network of half adders from the gate netlist of an addition circuit. Once the arithmetic bit-level representation of the circuit is obtained, equivalence checking can be performed using simple arithmetic operations. We have successfully applied the technique for the verification of a large number of multipliers of different architectures as well as more general arithmetic circuits, such as multiply/add units. The experimental results show the great promise of our approach.**

***Index Terms*—Arithmetic bit level, arithmetic circuit, datapath verification, equivalence checking, formal hardware verification, multiplier.**

## I. Introduction

In RECENT years, implementation verification by equivalence checking has become widely accepted. Modern equivalence checkers can handle circuits with hundreds of thousands of gates and have replaced gate-level simulation in many design flows. Equivalence checkers can perform extremely well if the two designs to be compared contain a high degree of structural similarity. This is usually the case after a conventional synthesis flow. Similarity means that the two circuits contain a lot of internal equivalences [1], [2], also called internal cut points [3]. Techniques to exploit these similarities have enabled equivalence checkers to verify huge combinational circuits, as has been shown by several authors [1]–[6]. On the other hand, if no internal equivalences exist, modern equivalence checkers fail and even for relatively small examples verification can become impossible.

One of the main problems encountered with equivalence checking in industrial practice is the inability to verify integer multipliers. The problem occurs when a register transfer level (RT-level) description of a circuit must be compared against a gate-level description. Typically, the latter has been generated from the former by some synthesis tool and it is the task of the equivalence checker to verify this synthesis process. The equivalence checker attempts to solve the problem by synthesizing a gate-level model from the RT model and by comparing the two gate-level designs. Unfortunately, this will fail in most cases. The problem is that the gate-level model generated by the equivalence checker looks entirely different, compared with the multiplier produced by the synthesis tool. Commercial equivalence checkers offer solutions for black-boxing multipliers; however, this and related solutions are cumbersome and may easily lead to false negatives.

Several approaches for multiplier verification can be considered. *Word-level decision diagrams* like binary moment diagrams (BMDs) [7] have great promise because they can efficiently represent integer multiplication. However, they require word-level information about a design, which is often not available and difficult to extract from a given bit-level implementation. Solutions based on *bit-level decision diagrams* such as binary decision diagrams (BDDs), e.g., as in [4] and [8], suffer from high complexity and may lack robustness, even if the BDDs are not built for the circuit outputs directly, but certain properties of the arithmetic circuits (e.g., "structural dependence" [8]) are exploited.

An approach based on a standard equivalence checking engine was proposed by Fujita [9]. Some arithmetic functions such as multiplication have special properties, which can be expressed as *recurrence equations*. For the circuit to be verified, it is checked whether the corresponding recurrence equation is valid using a standard cut-point-based equivalence-checking engine. The equivalence-checking problems are recursively broken down into smaller problems by case splitting on operand bits. The major drawback of this interesting approach is that for the circuit to be checked, a recurrence equation must exist and it must be known. This hampers automation of the verification task.

A related approach has been pursued in [10] and [11]. It is based on using certain arithmetic relationships of the implemented circuit function and, like [9], applies case splits on operand bits. The method is very promising for certain classes of circuits and makes efficient use of the circuit structure to find the right set of operand bits and a useful order on these bits for performing the case splits. In some cases, a good order exists such that the subproblems created indeed exhibit a great deal of structural similarities, thereby significantly simplifying the verification task. However, for some architectures, such as Wallace tree, no such order exists so that the method is not robust in some common practical cases.

*Reverse engineering* could be considered as a pragmatic approach to multiplier verification. Since the number of possible architectures for a multiplier is limited, one may incorporate a variety of architectures in the front end of the equivalence checker and repeat the comparison for all of them. We have not experimented with this approach, but we believe that there are

Manuscript received April 25, 2003; revised July 31, 2003 and October 8, 2003.

The authors are with the Department of Electrical and Computer Engineering, University of Kaiserslautern, 67653 Kaiserslautern, Germany (e-mail: stoffel@ieee.org).

Digital Object Identifier 10.1109/TCAD.2004.826548

| $167 \cdot 239$ | $167 \cdot 239$ | $239 \cdot 167$ | $239 \cdot 167$ |
|---|---|---|---|
| 334 | 1503 | 239 | 1673 |
| 501 | 501 | 1434 | 1434 |
| 1503 | 334 | 1673 | 239 |
| 39913 | 39913 | 39913 | 39913 |

Fig. 1. Multiplication example (decimal numbers).

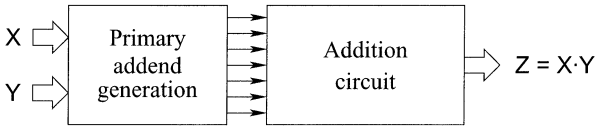


Fig. 2. Basic multiplier structure.

many obstacles. Note that even within one and the same architecture, e.g., a carry-save adder (CSA) array, there can be numerous implementation styles that have hardly any similarity in terms of internal equivalences. As an illustration, consider Fig. 1 showing four ways of multiplying two decimal numbers.

All four cases can be implemented by the same architectures, but have no internal equivalences at all. The adder stage of each row computes the accumulated sum of the previous rows. The accumulated sum values are different in all four variations. We experimentally verified the absence of internal equivalences using the $16 \times 16$ bit multiplier c6288. We modified the circuit by swapping its operands. Since multiplication is commutative c6288 with swapped operands must be equivalent to the original version. Proving this by our equivalence checker [12], however, turned out to be impossible. All internal equivalences were lost, except for the ones belonging to the partial products in the first circuit level.

In this paper, we propose a new approach to verification of arithmetic circuits. It can be understood as a reverse engineering process but at a more detailed level than described above. We propose an extraction technique which decomposes a gate netlist of an arithmetic circuit into its smallest arithmetic units. We do not identify word operations but bit operations and only consider the addition of single bits. Our extraction technique generates an arithmetic bit-level description of the circuit. Addition at this level is reduced to addition modulo 2 and generation of carry signals. The arithmetic bit level permits a very efficient verification algorithm.

Note that there have been a few approaches reported in literature based on identifying components of arithmetic circuits. Probably the earliest work related to the method described here is [13]. The technique is based on a logic-programming tool. It is similar to the one described in this paper in the respect that a higher-level representation of the circuit is extracted from a flat gate-level netlist. However, the gates in the circuit are mapped to component cells by syntactic pattern matching. The information about which cells [e.g., CSA or carry-lookahead adders (CLA) adders] have been used in the implementation must be given to the tool. The obtained representation is not a formal, generic arithmetic bit-level description as in our approach, but strongly depends on the cell types.

Identifying the components in a design can also be useful for a hierarchical verification approach. In [7], hierarchical verification was proposed for circuits that cannot be verified efficiently at the bit level (such as multipliers). For each component, a $*$BMD is constructed which can be compared against a specification for the component. Then, the $*$BMDs of the components are composed and the final $*$BMD is compared to the overall circuit specification. In another approach [14], a $*$PHDD is constructed for the circuit in a backward circuit traversal, similar to Hamaguchi's method [15]. The boundaries of the components are used as cut lines. As the cut is moved backward toward the primary inputs, the local Boolean functions between the cut lines are composed into the $*$PHDD. The components themselves are identified by BDD-based symbolic simulation.

Note that the component identification techniques proposed in [13] and [14] may work if the gate netlists are of highly regular structure and consist of exactly those components that are anticipated in the identification procedures. Unfortunately, this is not always the case for industrial multipliers as produced by modern synthesis tools which include various optimizations. Sometimes, even secondary utilities, such as netlist converters, can destroy the regularity of the circuit structure so that component identification techniques as in [13] or [14] face severe problems.

In contrast, the technique we propose in this paper is much more robust, with respect to the circuits it can handle. It does not require additional information about the cells from which the implementation is constructed. All building blocks are automatically decomposed into atomic addition operations by employing Boolean reasoning techniques on the gate netlist. In general terms, the proposed approach can be summarized as follows.

1) Decompose the two combinational circuits—where possible—into networks of one-bit addition primitives, such as XOR, half adder, full adder (arithmetic bit level).
2) Prove equivalence of corresponding circuit outputs on the arithmetic bit level using commutative and associative laws.

## II. Verification at the Arithmetic Bit Level

Arithmetic functions in digital circuits, such as addition, subtraction, multiplication, and division, are always implemented using addition as the base function. Subtracting a number $X$ in twos complement notation from a number $Y$, for example, is implemented by inverting all bits of $X$, adding 1, and adding $Y$. Multiplication is also based on addition. Hardware multipliers are most often composed of two stages (Fig. 2). In the first stage, the partial products are generated from two operand vectors, $X$ and $Y$. The way the partial products are generated depends on whether signed or unsigned numbers are processed, and whether or not Booth recoding is used. The partial products are inputs to the second stage which is an addition circuit. In the sequel we will call the inputs to an addition circuit *primary addends*. The addition circuit adds the primary addends up to produce the final result $Z = X \cdot Y$. The implementation of this addition circuit can be chosen from a variety of architectures differing in performance or area requirements. Most common implementations are an array of CSAs or a Wallace tree.

Note that this general structure consisting of a first stage generating primary addends and a second stage computing the arith-

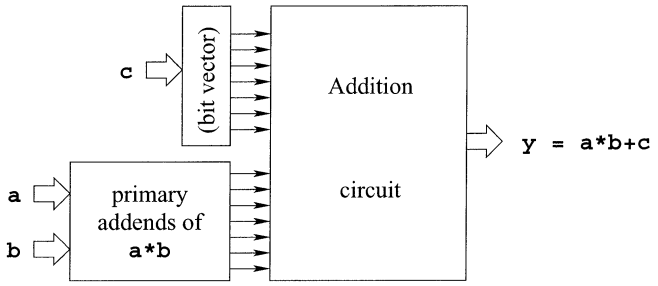


Fig. 3. More general arithmetic circuit example.

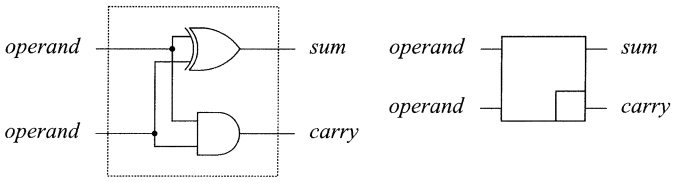


Fig. 4. Half adder, schematics, and symbol.

metic sum of these primary addends is not restricted to multipliers, but is found in many arithmetic circuits. Many applications require arithmetic RTL expressions comprised of addition, subtraction, and multiplication operators. Modern synthesis tools and module generators respond to this by offering more general building blocks such as multiply/add structures. Recently, commercial logic-synthesis tools also contain optimization algorithms specifically targeting such arithmetic expressions. Multiplication operators of the RTL code are decomposed into partial products and addition circuits. Then, all primary addends arising in the synthesized expression (including partial products and other addends) are accumulated in one addition circuit. This addition circuit can be constructed in a "globally" optimal structure, e.g., using a Wallace tree architecture. Fig. 3 shows an example of such a more general circuit structure generated from the RTL expression $\mathtt{y} = \mathtt{a} * \mathtt{b} + \mathtt{c}$. The primary addends for the addition circuit consist of the partial products from the multiplication $\mathtt{a}*\mathtt{b}$ and of the separate addend $\mathtt{c}$. As a consequence, the individual multiplication operators can no longer be identified in the generated netlist. Verification approaches relying on black-boxing of individual multipliers are bound to fail. Note that the approach proposed in this paper is insensitive to this kind of optimization since the general two-stage structure of primary addend generation followed by an addition circuit still prevails.

Any combinational circuit which performs the addition of binary bit vectors such as the addition stage in a multiplier can be represented as a composition of half and full adders. Fig. 4 shows the gate schematics of a half adder. In the sequel, we will use the half adder symbol shown on the right side of Fig. 4.

A full adder can be completely decomposed into half adders. We make use of this fact in our choice of arithmetic bit level representation. Fig. 5 shows a possible implementation of a full adder and the corresponding network composed of three half adders $P, Q, R$. Half adder $R$ adds the two carry bits $c_1$ and $c_2$ of the half adders $P$ and $Q$ and produces the full adder carry output $w$. Note that because the two signals $c_1$ and $c_2$ can never

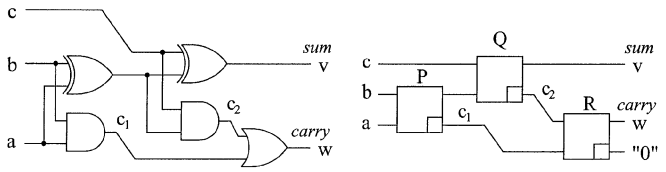


Fig. 5. Full adder decomposed into half adders.

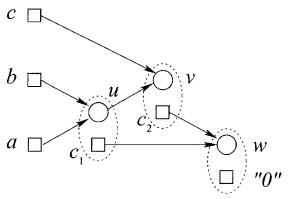


Fig. 6. Addition graph for full adder.

assume the logic value 1 simultaneously, the carry output of $R$ produces a constant 0.

Once we have obtained a representation of an addition circuit that is only composed of half adders, we speak of a *half adder network* or the *arithmetic bit level representation* of the circuit. This representation allows for a very efficient equivalence checking procedure. In the following, we introduce a mathematical model for the arithmetic bit level and develop the theoretical background of our verification procedure.

*Definition 1:* An *addition graph* is a triple $(G(V, E), R, F)$. $G(V, E)$ is a directed acyclic graph with vertex set $V$ and edge set $E$. The vertex set $V$ consists of three disjoint subsets, $V = S \cup C \cup I$. The vertices in $S$ have exactly two immediate predecessors, and are called *sum nodes*. The vertices in $I$ have no predecessors and are called *primary addends*. The vertices in $C$ have no predecessors and are called *carry nodes*. A vertex $v \in C \cup I$ is called *addend* of $G$.

$R$ is a relation, $R \subseteq (C \times S)$ and $F$ is a set of Boolean functions.

An addition graph $G$ is associated with a half adder network as follows. Each sum node in $G$ is associated with the sum output of a half adder in the network. Each carry node in $G$ is associated with the carry output of a half adder. Each primary addend in $G$ is associated with an input of the half adder network.

Two vertices $v$ and $w$ are connected by a directed edge $(v, w)$, if the half adder associated with $w$ has the signal associated with $v$ as operand.

For $c \in C$ and $s \in S$ it is $(c, s) \in R$ if and only if $c$ and $s$ are associated with the output signals of the same half adder in the network.

With each vertex $v \in V$ we associate the Boolean function $f_v \in F$ in terms of the primary addends that is implemented by the signal corresponding to $v$ in the half adder network. □

For illustration of this definition, Fig. 6 shows the addition graph of the full adder of Fig. 5. Note that the primary addends and the carry nodes are the source nodes of an addition graph, and are also referred to as addends in the following. In Fig. 6, addends are represented by boxes, sum nodes are represented by circles. The relation between carry and sum nodes is indicated by dashed lines. Nodes $v$ and $w$ are sinks of the addition graph and correspond to outputs $v$ and $w$ of the half adder network.

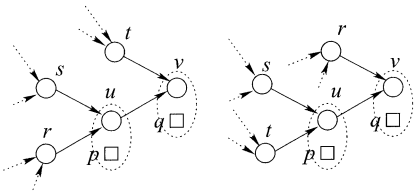


Fig. 7. Addition graph of Lemma 1.

The modeling of a half adder by two separate nodes in the addition graph may seem awkward. Note, however, that our definition leads to a decomposition of the half adder network into graph entities such that all but the source vertices correspond to XOR operations. Therefore, each sum node in the graph can be associated with the sum modulo 2 of all source nodes in its transitive fanin. This facilitates the manipulation of the graph structure.

In the following, without loss of generality, we assume that the addition graph is a forest of *trees*. If the addition graph obtained from the original half adder network does not have tree structure, we can always generate a forest of trees by duplication of appropriate graph portions including primary addends.

*Lemma 1:* Let $r$ and $s$ be the operands of a sum node $u$ in an addition graph. Further, let $u$ and $t$ be the operands of a sum node $v$, as shown in Fig. 7. Let $p$ and $q$ be the carry nodes of $u$ and $v$, respectively. Exchanging operand $r$ with operand $t$ does not change $f_v$ and does not change $f_p \oplus f_q$.

*Proof:* Function $f_v$ does not change because addition modulo 2 is commutative. The function $f_p \oplus f_q$ does not change, because $(r \cdot s) \oplus ((r \oplus s) \cdot t) = (t \cdot s) \oplus ((t \oplus s) \cdot r)$. ■

Half adder networks implementing practical addition stages have the special property that each addition tree computes a digit of a binary encoded integer. The carry signals of the addition tree for digit $i$ all feed into the addition tree for the next digit, $i+1$. This can be exploited when checking the equivalence of addition trees in practical addition networks.

*Lemma 2:* The output functions of two addition trees $T$ and $\tilde{T}$ (Fig. 8) are equivalent if the following conditions are true.

1) The sets of primary addends for $T$ and $\tilde{T}$ are identical $(I_T = I_{\tilde{T}})$.
2) There exists an addition tree $S$ such that the set of all carry nodes being addends for $T$ is identical with the set of carries generated in $S$. The same holds for $\tilde{T}$ and some addition tree $\tilde{S}$.
3) The output functions of $S$ and $\tilde{S}$ are equivalent.

*Proof:* If the output functions of $S$ and $\tilde{S}$ are equivalent, then the sum modulo 2 of all carries generated in $S$ is equivalent to the sum modulo 2 of all carries generated in $\tilde{S}$. This follows from the observation that $S$ can be transformed into $\tilde{S}$ by a sequence of operand swaps according to Lemma 1. $T$ as well as $\tilde{T}$ compute the modulo 2 sum of the primary addends and the carries of $S$. ■

Once we have a representation of an addition circuit as a half adder network, the equivalence check using Lemma 2 is straightforward. Note that finding addition tree $S$ for addition tree $T$ in condition 2 is trivial in practice, since $S$ is located in the immediate structural vicinity of $T$. The correspondences

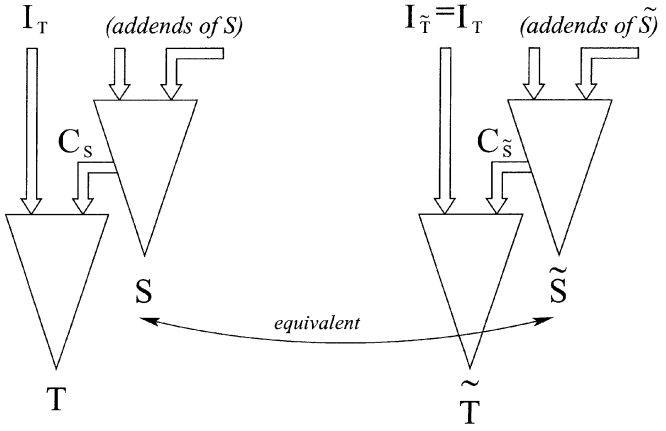


Fig. 8. Illustration of Lemma 2.

$\tilde{S}$ with $S$ and $\tilde{T}$ with $T$ are known from the given equivalence checking task.

Note the recursive nature of Lemma 2: the equivalence of the output digit $i$ (tree $T$) depends on the equivalence of digit $i-1$ (tree $S$). The terminal case of the recursion is digit 0 where no carry-ins exist and only condition 1 of the lemma needs to be checked. The total runtime of the equivalence check according to Lemma 2 is linear in the number of half adders which is proportional to circuit size.

Another possibility to verify addition circuits on the arithmetic bit level is to manipulate the circuits using the operation of Lemma 1 until both circuits have the same structure and contain enough internal equivalences for a standard equivalence checking procedure to be successful.

The problem that remains to be solved, however, is how to extract the arithmetic bit-level representation from the gate netlist of an addition circuit. This is the subject of the following section.

## III. Extracting the Half Adder Network

An addition circuit can be implemented in many different ways. Different architectures, e.g., carry-save adder arrays or Wallace trees, exist, aiming at different design goals. Also for the components and subcomponents, there exists a variety of implementation choices. As an example of an adder stage which is not constructed from cascaded half and full adders, consider the four-bit carry-lookahead adder of Fig. 9. In order to speed up computation time, the carry signals in each output cone are generated by a special logic block.

It is our goal to extract a half adder network that abstracts from such implementation details. We seek an extraction technique that produces as output a network of half adders which is functionally equivalent to the implementation.

### A. Basic Procedure

The approach we propose is based on the following assumption. The predominant operation at the bit level is the computation of exclusive OR. This logic function is part of every implementation of binary addition. We use Boolean reasoning techniques [12] to detect XOR relationships in the original circuit. Note that there are many possibilities to implement XOR

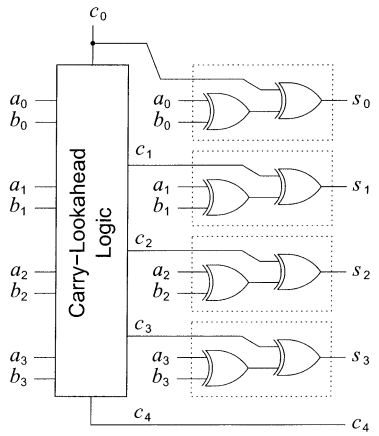


Fig. 9. Four-bit carry-lookahead adder.

detection algorithms, e.g., using SAT, local BDDs, or structural hashing techniques. In order to trade off performance against quality of results, it is desirable to have several phases with different algorithms. Although we have not experimented with this, we believe that, as a first extraction phase, a structure-based functional hashing technique, e.g., the technique based on AND/INVERTER graphs of [16], could be quite efficient to extract the majority of the XOR functions in an arithmetic circuit. The few remaining XOR functions could then be detected by a more powerful yet more time-consuming functional analysis based on SAT, BDDs, or automatic test pattern generation (ATPG).

Guided by the detected XORS, we construct a network of half adders as a reference circuit. We store implications between nodes in the original circuit and the half adder network. The stored implications establish a mapping between the nodes of the original and the reference circuit.

As an example, consider the implementation of a full adder as shown in Fig. 10.

Using Boolean reasoning techniques it is possible to prove that the signal $x$ can be expressed as the exclusive OR of signals $a$ and $b$. As a consequence, in the reference circuit, we insert a half adder node $u$ with operands $a$ and $b$ and store implications reflecting the equivalence of the sum output of the half adder and node $x$. Also, signal $p$ can be expressed as the exclusive OR of $x$ and $c$. We insert a half adder node $v$ with operands $x$ and $c$ and store the equivalence of the sum output with signal $p$.

Now that the half adders $u$ and $v$ exist, it is possible to express signal $q$ as an exclusive OR of the carry outputs $c_1$ of $u$ and $c_2$ of $v$. Also, we can identify the implication $c_1 = 1 \rightarrow c_2 = 0$, which is equivalent to $c_1 \cdot c_2 = 0$, for all possible input vectors of the adder circuit. Therefore we insert half adder $w$ with operands $c_1$ and $c_2$, and we store the information that the carry output of this half adder produces a constant 0. We also store an equivalence pointer between the sum output of $w$ and the output $q$ of the adder circuit. We now have a complete mapping of the adder circuit as a half adder network.

Note that although function $q$ implements the majority function, $q = (a + b)c + ab = ab + ac + bc$, of the inputs $a, b, c$ and not an XOR function of any of these operands, we can still find a mapping for this node by using signals from the reference circuit.

When detecting an XOR relationship of the form $y = a \oplus b$ for some signal $y$ in the original circuit, with $a$ and $b$ being signals in the original or in the reference circuit, it is actually not sufficient to insert a half adder with operands $a$ and $b$. It could be that an operand has to be inverted in order to make the half adder useful as an operand later. Since the correct operand phases cannot be determined by the XOR detection $(y = a \oplus b = \bar{a} \oplus \bar{b})$, we add not only one half adder for each XOR found but all four half adders corresponding to the four possible combinations of inversions of the operands.

The pseudocode of Table I showing subroutine extraction_pass $(C, R, O)$ summarizes the basic procedure just described. It performs one sweep over the implementation circuit $C$ to detect XOR relationships and to construct a reference circuit $R$ by inserting the corresponding half adders. By setting the parameter $O$ to either $C$ or $R$, the algorithm is controlled to identify XORs in the design or in the reference circuit, respectively. This basic routine is being used several times in the overall algorithm to be described in Section III-C.

The Boolean analysis underlying the presented procedure is local and of fairly low complexity. The reason is that the reference circuit is constructed step by step guided by the XOR functions found in the implementation circuit. The reference circuit and the implementation exhibit a great amount of structural similarity which is represented by equivalence pointers, so that new XOR relationships can be proven by using the equivalence pointers as "reasoning short-cuts."

### B. Local Half Adder Network Extensions

In practical implementations, the calculation of sum and carry signals may be locally separated and restructured, e.g., to improve timing. If such local optimizations have been performed, the basic procedure of Section III-A may not always be sufficient to determine a complete mapping of the circuit. However, since the internal nodes of our addition trees represent only XOR functions, reverse-engineering these trees using commutative and associative transformations is simple. We analyze the current structure of the reference circuit and locally add promising new half adders. Then we target the unmapped nodes again as described in Section III-A using the new half adders as potential XOR operands.

Table II shows pseudocode of the steps taken in this procedure. It is explained using a small example. Consider the piece of circuitry in Fig. 11. It can be viewed as part of a larger addition circuit. Signal $u$ computes the sum of the signals $a, b, c, d$, and $e$. The sum is computed as a tree which is balanced appropriately in order to minimize delays. Signal $v$ computes the corresponding carry. For simplicity, all XOR functions in this circuit are implemented by XOR gates so that they can be easily identified by the reader.

The basic procedure described in Section III-A constructs a half adder network for this gate netlist. The result is shown in Fig. 12. The individual steps in the construction of this reference circuit are as follows. Procedure extraction_pass $(C, R, C)$ is called with parameters set such that XORs are searched within circuit $C$. For the XORs found, it inserts corresponding half adders in $R$ and stores equivalence pointers

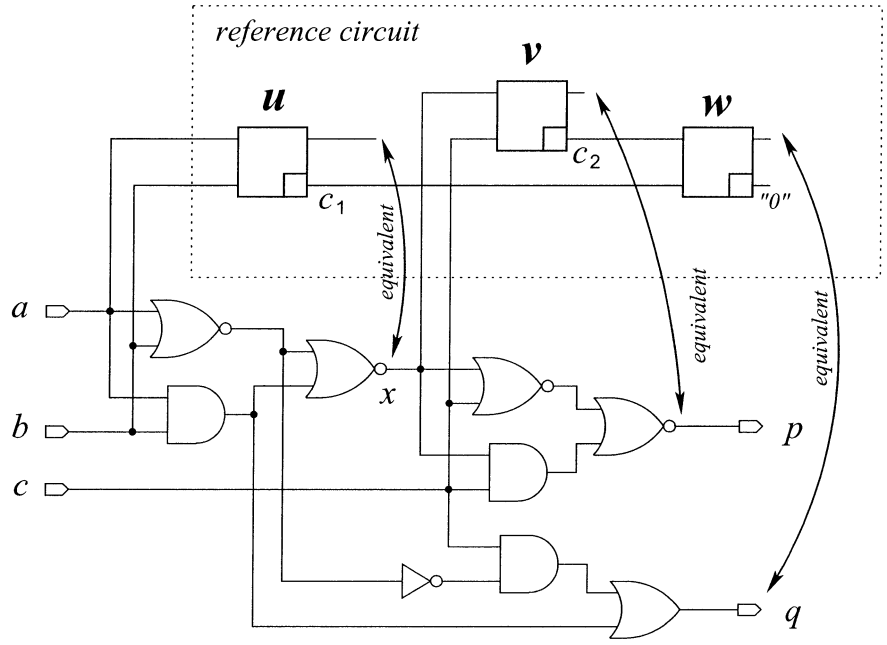


Fig. 10. Full adder implementation and mapped half adder network (reference circuit).

TABLE I
SUBROUTINE PERFORMING A HALF ADDER EXTRACTION PASS

```
extraction_pass(C, R, O)
{
  /* C: original gate netlist */
  /* R: reference circuit */
  /* O: circuit to choose XOR operands from */
  for all unmapped nodes y in C {
    while (exist a, b ∈ O with y = a ⊕ b) {
      insert corresponding half adders H_i in R;
      store equivalences of half adder sums with y;
      mark nodes in C covered by H_i as mapped;
  }}
}
```

TABLE II
EXTENDED PROCEDURE TO MAP UNMAPPED CIRCUITRY

```
map_gap(C, R, k)
{
  /* C: original gate netlist */
  /* R: reference circuit */
  /* k: output signal of unmapped circuit region, k ∈ C */
  backtrace in C from k
    until signals mapped to HA sum outputs
    or primary addends are reached:
    I := reached signals;  /* gap inputs */
  G := subcircuit with inputs I and output k; /* gap */
  A := ∅; /* set of addends */
  for each gap input j in I {
    backtrace from j in R until addends are reached:
      J := reached signals;  /* addends of j */
    A := A ∪ J;
  }
  forward trace from signals in A
    until signal computing sum of addends in A is reached;
    u := reached signal;
  while untried configurations exist {
    /* produce new configuration of HAs */
    insert HA configuration to produce sum u based on signals I;
    /* extract HAs on unmapped circuit region: */
    extraction_pass(G, R, R);  /* see Table I */
    if signal k mapped
      return;
    discard inserted HAs;
  }
}
```

between the half adder outputs in $R$ and corresponding gates in the implementation $C$. Signals with equivalent functions in Figs. 11 and 12 are given the same names. For the XORs producing the signals $g, h, u, f, j$, and $v$ it inserts the half adders $A, B, C, D, E$, and $F$, respectively. In a second procedure call, extraction_pass $(C, R, R)$, XORs are searched in terms of signals in the reference circuit. Now, signal $l$ is identified to be expressible as an XOR of the carry outputs of the half adders $D$ and $E$. This XOR leads to the insertion of half adder $G$.

Due to the local optimizations that have been done in the circuit after separating sum and carry computation, we obtain an incomplete mapping. On one hand, the half adders $A, B$, and $C$ produce carry signals which are not connected. On the other hand, we have a "gap" in the network for which no half adders have been inserted. Signal $k$ which is used as an operand to half adder $F$ cannot be mapped to the output of any half adder. Obviously, the facts that some half adder carry outputs are not used (see question marks in Fig. 12) and that there is unmapped circuitry ("gap" in Fig. 12) are closely related.

In order to overcome such gaps in the reference circuit, we need to "undo" the optimizations that have lead to an incomplete mapping. We need to find a configuration of half adders for the sum signal $u$ such that the unmapped circuitry can be expressed in the carry signals of these half adders. This can be done in the following way (Table II).

First, by a backtrace procedure, we identify the inputs of the gap. These are signals that are either primary addends or signals mapped to half adder sum outputs in the reference circuit. In our example, by backtracing from signal $k$, we identify signals $j, b$, and $c$ as inputs to the gap. Under the assumption that the gap function computes the carry corresponding to the sum of

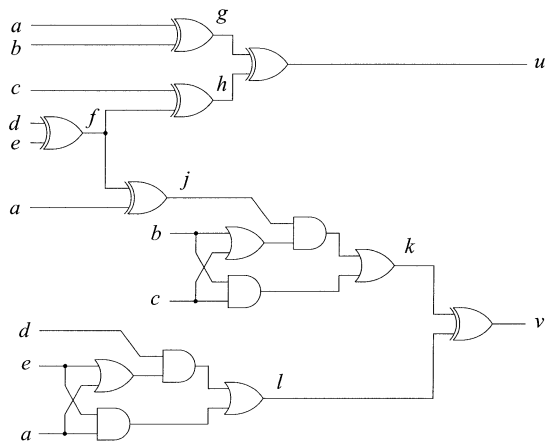


Fig. 11. Optimized circuitry.

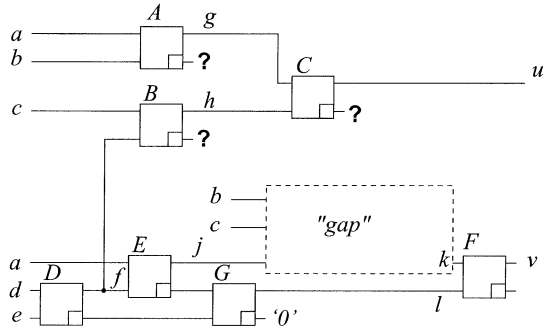


Fig. 12. Reference circuit with "gaps."

signals $j, b$, and $c$, we search in the reference circuit a signal that computes this sum. This analysis is easy in the addition graph (Definition 1) corresponding to the reference circuit. By backtracing from the gap inputs, we determine all addends, i.e., primary addends and carry nodes, contributing to the sum. In our example, the gap inputs $b$ and $c$ are primary addends. Gap input $j$ computes the sum of $a, d$, and $e$. Hence, we search for a signal that computes the sum of the addends $a, b, c, d$, and $e$. We identify this signal by a forward trace in the addition graph starting at the addends. In our example, we identify signal $u$.

Note that the half adders computing the sum $u$ do not make use of signal $j$. Therefore, we now locally extend the reference circuit by inserting a different configuration of half adders such that signal $j$ and all other addends being inputs to the gap are used. One possible configuration is shown in Fig. 13. After inserting the half adders $H$ and $J$ in this way, we have obtained an alternative representation of signal $u$ that makes use of the signals at the inputs of the gap. Now by calling extraction_pass $(G, R, R)$ the problematic circuitry is revisited. It determines that signal $k$ in Fig. 13 can be expressed as an XOR of the carries produced by half adders $H$ and $J$. This leads to an additional half adder $K$ completing the mapping for the circuitry in the gap. The final result is shown in Fig. 14. We have, in effect, reversed the optimizations in the netlist and we can discard the half adders $A, B$, and $C$. The resulting reference circuit is a coherent network of half adders.

In this example, there are three possible configurations of half adders to sum up signals $j, b$, and $c$ in Fig. 13. In this case, all

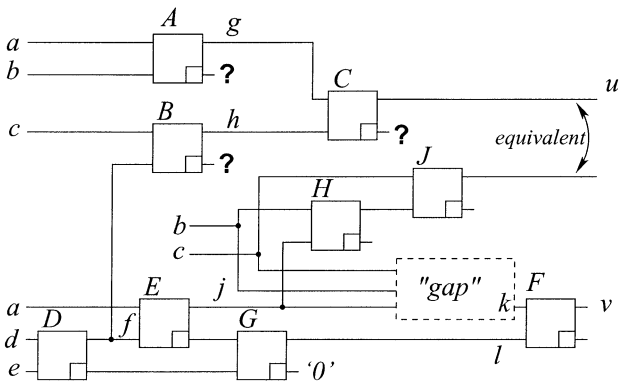


Fig. 13. Local extension of the half adder network'.

three network extensions lead to a successful mapping of the gap. In general, this may not always be the case and all possible configurations have to be tried until the correct mapping is found. However, since this type of optimization happens very locally, the computational effort to find the right network extension is usually very low.

Note that there are many ways to optimize the building blocks of addition circuits [17]. Some optimizations as, e.g., a carry-select adder cannot be handled by the procedure as it is explained in this section. In order to cope with such structures, further extensions to the approach would be necessary. However, we have not found optimizations of this kind in our experiments.

Other optimizations are handled naturally by our approach, including the carry-lookahead architecture. Carry-lookahead adders are often employed for the final adder in a multiplier. Fortunately, this can be handled by our basic procedure of Section III-A without any extensions. Consider Fig. 9 again. The Boolean reasoning technique detects XOR relationships for every carry signal $c_1$ to $c_4$, each leading to a new half adder inserted in the reference circuit. The extracted half adders, in fact, form a "ripple-carry" structure. Also in this case, the extraction procedure reverses the optimizations of the netlist.

### C. Algorithm

Table III shows the pseudocode of the proposed algorithm for half adder network extraction. The algorithm consists of four phases. The first two phases consist of the steps introduced in the example of Fig. 10. The third phase targets the remaining unmapped nodes as described in Section III-B. In each of these phases, subroutine extraction_pass() shown in Table I is called which performs one pass over the original circuit, analyzing whether XOR relationships exist for every node that has not been mapped by a half adder yet. Depending on the phase, the XOR operands are searched either in the original or in the reference circuit. Finally, in the last phase, a backtrace procedure is started to collect a set of half adders forming a cover for the given addition circuit. This cover is used for the equivalence check of Section II.

In the case that the arithmetic circuit processes signed numbers, e.g., signed multiplication or subtraction is performed or a recoding technique such as Booth recoding is used, special consideration has to be taken when assembling the half adders to a coherent network. In this case, some of the primary addends

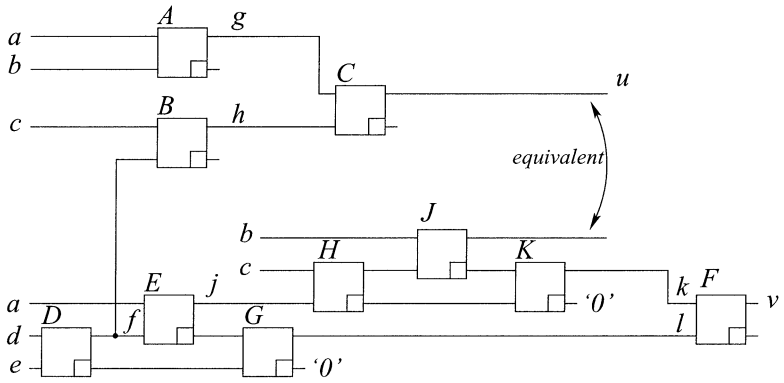


Fig. 14. Successful half adder extraction on unmapped circuitry.

TABLE III
ALGORITHM FOR HALF ADDER NETWORK EXTRACTION

```
extract_half_adder_network(C)
{
  /* input: C, original gate netlist */
  R := ∅;    /* reference circuit */

  /* STEP 1: search for XORs in original circuit, C */
  extraction_pass(C, R, C);
  /* STEP 2: search for XORs in reference circuit, R */
  extraction_pass(C, R, R);
  /* STEP 3: complete mapping for yet unmapped nodes */
  for all unmapped regions y in C {
    /* locally extend half adder network R for y; */
    map_gap(C, R, y);
  }
  extraction_pass(C, R, R);
  /* STEP 4: find cover */
  foreach output y of circuit C {
    /* DFS backtrace in half adder network */
    select a half adder h mapped on y;
    push h on stack;
    while stack not empty {
      pop half adder h from stack; mark h;
      foreach operand o of h {
        select a half adder i mapped on o;
        push i on stack;
  }}}
  remove unmarked half adders;
  return R;
}
```

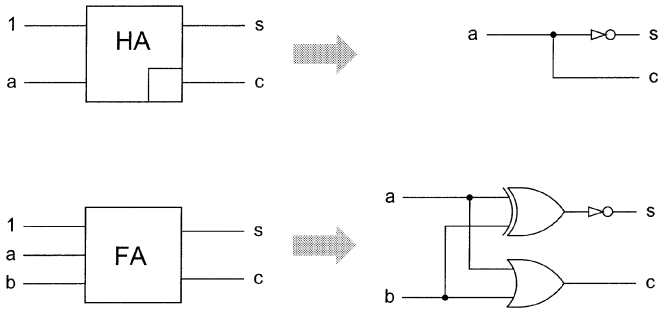


Fig. 15. Logic simplification due to constant addends.

may be subtracted instead of added. In two's complement representation this amounts to logically inverting the individual bits and adding a constant 1 into the least significant addition tree. In a similar way, often, sign extension of addends in addition trees is done by inverting the most significant bit (MSB) of the addend and adding constant 1's into all bit positions from the MSB up to the required bit width.

When extracting half adder networks in these cases, we have to keep two things in mind. First, some of the primary addends may have to be added in logically inverted phase. Second, constant 1's may have been added into an addition tree. These constants simplify the logic. Fig. 15 shows the effect of a constant addend of 1 to a half adder and to a full adder, respectively. A half adder with one operand being constant 1 degenerates to a simple fanout system with one inverting and one noninverting branch. A full adder with an operand of constant 1 degenerates to an inverted XOR gate and an OR gate. This has consequences for the final mapping stage of Table III as well as for the half adder extraction algorithm of Table I which relies entirely on functional (Boolean) analysis to identify XORs. These algorithms have to be extended in the following way. In addition trees with possible constant addends of 1 any XOR found in the circuit can be mapped not only to a half adder but also to a degenerated full adder as in Fig. 15. This has to be accounted for when generating the reference circuit. Also, this increases the search space for the final mapping stage.

Half adders with constant operands degenerating to simple fanout systems as in Fig. 15 cannot be identified by XOR detection. However, because of their simple structure they can be easily identified through their carry outputs. Since the carry signal is functionally equivalent to the nonconstant addend of the half adder, this addend appears twice: in two addition trees of successive output bit significance. In the final mapping stage, these situations are identified and accounted for by a constant 1 added to the less significant of the two addition trees.

Note that our procedure is robust also in cases where the basic building blocks of the arithmetic circuit are not half or full adders. This has already been illustrated for carry-lookahead adders in Section III-B.

## IV. VERIFICATION FRAMEWORK

As discussed in Section I, arithmetic circuits pose great problems to current equivalence checkers. In order to circumvent these problems, most commercial tools provide mechanisms to black-block multipliers embedded in the designs so that at least the remaining circuitry can be formally verified. Ideally, the black-boxed multipliers can be checked using special verifica-

TABLE IV
ARITHMETIC BIT-LEVEL EXTRACTION ON BOOTH-ENCODED MULTIPLIER GATE NETLISTS

| circuit origin | signed/ unsigned | archi- tecture | bit vector widths $X$ | $Y$ | $Z$ | size (# conns.) | CPU time (h:mm:ss) |
|---|---|---|---|---|---|---|---|
| comm. 2 | unsigned | wall | 4 | 4 | 08 | 185 | 0:00:00 |
| comm. 1 | unsigned | csa | 6 | 8 | 14 | 459 | 0:00:02 |
| comm. 1 | unsigned | wall | 6 | 8 | 14 | 500 | 0:00:02 |
| comm. 1 | unsigned | wall | 8 | 6 | 14 | 504 | 0:00:02 |
| comm. 1 | signed | wall | 8 | 8 | 16 | 585 | 0:00:03 |
| comm. 2 | unsigned | wall | 8 | 8 | 16 | 648 | 0:00:03 |
| comm. 2 | unsigned | wall | 12 | 12 | 24 | 1404 | 0:00:09 |
| comm. 1 | signed | wall | 22 | 6 | 28 | 1393 | 0:00:08 |
| comm. 1 | unsigned | wall | 22 | 6 | 28 | 1424 | 0:00:07 |
| comm. 1 | unsigned | wall | 6 | 22 | 28 | 1326 | 0:00:09 |
| comm. 1 | signed | wall | 16 | 16 | 32 | 2324 | 0:00:29 |
| comm. 2 | signed | wall | 16 | 16 | 32 | 2340 | 0:00:26 |
| comm. 2 | unsigned | wall | 16 | 16 | 32 | 2443 | 0:00:20 |
| comm. 1 | signed | wall | 15 | 22 | 37 | 2955 | 0:00:42 |
| comm. 1 | signed | wall | 22 | 15 | 37 | 3052 | 0:00:54 |
| comm. 1 | unsigned | wall | 15 | 22 | 37 | 2944 | 0:00:49 |
| comm. 1 | unsigned | wall | 22 | 15 | 37 | 3174 | 0:00:53 |
| comm. 1 | unsigned | wall | 26 | 16 | 42 | 3891 | 0:01:02 |
| comm. 2 | signed | wall | 24 | 24 | 48 | 5178 | 0:01:32 |
| comm. 2 | unsigned | wall | 24 | 24 | 48 | 5319 | 0:01:27 |
| comm. 2 | unsigned | wall | 32 | 32 | 64 | 9265 | 0:04:34 |
| comm. 2 | signed | wall | 48 | 48 | 96 | 20203 | 0:29:50 |
| comm. 2 | unsigned | wall | 48 | 48 | 96 | 20464 | 0:22:09 |

tion techniques. In our approach, we assume such a set-up. The verification technique proposed in this paper can be directly applied to the black-boxed arithmetic circuitry.

After the arithmetic circuitry has been black-boxed, information about the encoding of the partial product bits is required. Fortunately, the number of encodings used in practice is small. In our experiments, the encoding was given to the verification program in the form of specification circuitry for the partial products. The extraction procedure maps signals of the implementation to XORs expressed in signals of the specification, using Boolean reasoning techniques. If successful, this formally verifies the correct encoding.

In a practical application where the information about the encoding is not available, it may be desirable to automatically determine the encoding. A straightforward approach is to simply explore all possibilities, starting with the most probable ones. This need not be time consuming, since the number of possible encodings is rather small and the wrong encodings are filtered out quickly. (The extraction procedure is aborted in an early phase if there are no primary addends to map onto.)

Although black-boxing of arithmetic circuits is standard practice, it is still interesting to ask how the approach proposed in this paper could be integrated directly as an additional heuristic into existing equivalence checking frameworks. We have not experimented with this, but the following scenario of an integrated procedure seems promising. Standard similarity-based equivalence checking is run for the given circuits in the usual way. If the circuits contain multipliers (or other arithmetic blocks), the procedure is likely to fail. Let us assume that the front end of the equivalence checker has generated a specification with the same encoding of the partial product bits as in the implementation. Then, the standard techniques advance the cut frontier up to and including these partial product bits. In the addition network that follows, however, the procedure is likely to fail by lack of similarity. Eventually, the equivalence checker aborts the process.

This is the point where the proposed extraction technique can be invoked. Whenever the equivalence checker starts "choking," we need to identify the partial product bits for every arithmetic block in the implementation. This is easy because we know the partial products in the specification and the standard equivalence checking process has already identified their equivalent counterparts in the implementation. Knowing the partial products in the implementation we can examine whether or not the cut frontier has successfully traversed the corresponding addition network. If not, the extraction procedure is started and equivalence is proved for the outputs of the arithmetic blocks. Once this is completed, standard equivalence checking can continue making use of these new cut points.

In order to ensure that the standard equivalence checker can match the partial product bits of the implementation with the partial product bits of the specification, the front end of the equivalence checker should generate all common versions of partial product bits as candidates for cut points in the standard procedure. As already discussed above, in our experience, the number of different encodings being used in industrial practice is fairly small so that the overhead for generating all the candidates should be reasonable.

Note that the proposed extraction procedure will fail to extract an arithmetic bit-level description if the multiplier circuit contains an error. This, however, is easily detected by a simulation step earlier in the verification flow. Experience from ATPG, e.g., [18], shows that multipliers are highly random-pattern testable so that a buggy design is usually detected by only a small number of random patterns.

TABLE V
ARITHMETIC BIT-LEVEL EXTRACTION ON NON-BOOTH-ENCODED MULTIPLIER GATE NETLISTS

| circuit origin | signed/ unsigned | archi- tecture | bit vector widths $X$ | $Y$ | $Z$ | size (# conns.) | CPU time (h:mm:ss) |
|---|---|---|---|---|---|---|---|
| comm. 2 | unsigned | wall-bk | 4 | 4 | 08 | 110 | 0:00:00 |
| comm. 2 | unsigned | csa | 4 | 4 | 08 | 110 | 0:00:00 |
| comm. 2 | unsigned | wall-rc | 4 | 4 | 08 | 104 | 0:00:00 |
| comm. 1 | unsigned | csa | 6 | 8 | 14 | 472 | 0:00:02 |
| comm. 1 | unsigned | csa | 8 | 6 | 14 | 437 | 0:00:02 |
| comm. 1 | unsigned | wall | 6 | 8 | 14 | 425 | 0:00:02 |
| comm. 1 | unsigned | wall | 8 | 6 | 14 | 425 | 0:00:03 |
| comm. 1 | unsigned | csa | 8 | 8 | 16 | 640 | 0:00:02 |
| comm. 1 | unsigned | wall | 8 | 8 | 16 | 604 | 0:00:03 |
| comm. 2 | unsigned | wall | 8 | 8 | 16 | 608 | 0:00:03 |
| comm. 2 | unsigned | wall-rc | 8 | 8 | 16 | 528 | 0:00:02 |
| comm. 2 | signed | wall | 12 | 12 | 24 | 1502 | 0:00:37 |
| comm. 2 | unsigned | wall | 12 | 12 | 24 | 1478 | 0:00:13 |
| comm. 2 | unsigned | wall-rc | 12 | 12 | 24 | 1272 | 0:00:08 |
| comm. 1 | signed | wall | 22 | 6 | 28 | 1372 | 0:00:22 |
| comm. 1 | signed | wall | 6 | 22 | 28 | 1394 | 0:00:27 |
| comm. 1 | unsigned | csa | 22 | 6 | 28 | 1254 | 0:00:08 |
| comm. 1 | unsigned | csa | 6 | 22 | 28 | 1499 | 0:00:10 |
| comm. 1 | unsigned | wall | 22 | 6 | 28 | 1362 | 0:00:11 |
| comm. 1 | unsigned | csa | 16 | 16 | 32 | 2682 | 0:00:44 |
| comm. 1 | unsigned | wall | 16 | 16 | 32 | 2778 | 0:01:37 |
| comm. 2 | unsigned | wall | 16 | 16 | 32 | 2740 | 0:00:38 |
| comm. 2 | unsigned | wall-rc | 16 | 16 | 32 | 2336 | 0:00:22 |
| c6288 | unsigned | csa | 16 | 16 | 32 | 5568 | 0:00:33 |
| c6288nr | unsigned | csa | 16 | 16 | 32 | 4698 | 0:01:37 |
| c6288opt | unsigned | csa | 16 | 16 | 32 | 4721 | 0:00:21 |
| comm. 1 | signed | wall | 15 | 22 | 37 | 3652 | 0:02:14 |
| comm. 1 | unsigned | csa | 15 | 22 | 37 | 3541 | 0:01:23 |
| comm. 1 | unsigned | csa | 22 | 15 | 37 | 3524 | 0:01:14 |
| comm. 1 | unsigned | wall | 15 | 22 | 37 | 3652 | 0:01:54 |
| comm. 1 | unsigned | csa | 16 | 26 | 42 | 4532 | 0:01:51 |
| comm. 1 | unsigned | csa | 26 | 16 | 42 | 4524 | 0:01:47 |
| comm. 1 | unsigned | wall | 16 | 26 | 42 | 4713 | 0:03:56 |
| comm. 1 | unsigned | wall | 26 | 16 | 42 | 4672 | 0:03:39 |
| comm. 2 | unsigned | wall | 24 | 24 | 48 | 6410 | 0:03:40 |
| comm. 2 | unsigned | wall-rc | 24 | 24 | 48 | 5424 | 0:01:49 |
| comm. 2 | signed | wall | 32 | 32 | 64 | 11636 | 0:24:13 |
| comm. 2 | unsigned | wall | 32 | 32 | 64 | 11612 | 0:13:35 |
| comm. 2 | unsigned | wall-rc | 32 | 32 | 64 | 9792 | 0:05:48 |
| comm. 2 | unsigned | wall-rc | 48 | 48 | 96 | 22368 | 0:29:21 |

If it is desirable to represent the arithmetic circuit by a word-level decision diagram, our approach can also be of interest. It was already pointed out in [7], [14], and [19] that knowledge about the subcomponents of a multiplier can be useful in BMD construction. It seems likely that the arithmetic bit-level representation as extracted by the procedure of Section III could be a good basis for heuristically guiding a BMD construction process along the lines of [14] and [15].

## V. EXPERIMENTAL RESULTS

The described techniques have been implemented as a part of the HANNIBAL [12] tool. Tables IV and V show some of our results for extracting the half adder networks for multipliers of different origin, bit widths, and architectures. They were generated by commercial tools and many of them were extracted from real designs. The examples reflect the architectures currently used in industrial practice. In our experiments, we found that for multipliers generated by our own self-written generators, it was much easier to extract the arithmetic bit-level representation than for circuits generated by commercial synthesis tools. These circuits are much more "sophisticated" than their academic counterparts, which are constructed using common textbook components and architectures. Therefore, we do not report verification results for circuits generated by our own tools.

Tables IV and V list multipliers with and without Booth encoding, respectively. In each table, the first column shows the origin of the circuit. We have experimented with multipliers generated by two different commercial tools, one of them being Synopsys Design Compiler (labeled "comm. 1"). Additionally, we have three different versions of C6288 from the well-known ISCAS'85 benchmark set in the table. Circuit c6288 is the original circuit, circuit c6288nr is its nonredundant version, and circuit c6288opt is the result of optimizing c6288 using sequential interactive synthesis (SIS) with script.rugged.

The various multipliers process signed or unsigned numbers (column 2). The multiplier architectures are given in column 3: "wall" means Wallace tree, "csa" is CSA array type. The multipliers labeled "wall-rc" and "wall-bk" are Wallace-tree-type

TABLE VI
ARITHMETIC BIT-LEVEL EXTRACTION ON EXPRESSIONS

| circuit | # mult.'s | result bit width | # conn.'s | # CPU time (s) |
|---|---|---|---|---|
| MAC1 | 2 | 17 | 1528 | 7 |
| MAC8 | 9 | 17 | 12034 | 95 |

multipliers with a final adder of ripple-carry or Brent-Kung type [20], respectively. Columns 4–6 show the bit widths of the multiplication operands, $X, Y$, and the result, $Z$. Column 7 shows the circuit size given as the number of connections in the netlist. The last column reports on the CPU time on a 1300 MHz PC running Linux.

The benchmarks differ greatly with respect to the architectures used, such as Wallace trees and arrays of RCA or CSA adders. For all these architectures, the arithmetic bit level could be extracted within short CPU times. Note that, due to the Boolean nature of our extraction technique, the arithmetic bit level can also be obtained if the multiplier has been been optimized using standard logic synthesis techniques. This is illustrated by means of c6288opt and logic synthesis by SIS. The CPU times needed for the extraction experiments are not in direct proportion to the circuit sizes. The main reason for this is that the circuits may contain different optimizations such that the extended extraction methods as described in Section III-B have to be invoked differently often. Compare, for example, the extraction times for the signed and unsigned $32 \times 32$-bit multipliers in Table V. Although both circuits are of roughly the same size, extracting the signed multiplier takes almost twice as long as extracting the unsigned version.

We verified the equivalence between any pair of multipliers with the same operand widths and number interpretation (signed/unsigned) using the equivalence check of Lemma 2. After the arithmetic bit level was extracted, the actual equivalence check in all cases took only a fraction of a second. In the cases where a different encoding was used (Booth versus non-Booth), each of the circuits was checked against the reference circuit with the same encoding produced by the front end of the equivalence checker.

In a second experiment (Table VI), we ran the arithmetic bit-level extraction algorithm on circuits computing larger arithmetic expressions containing several multipliers. These netlists were generated by Synopsys DC Ultra, which features advanced arithmetic optimization capabilities as described in Section II by identifying and merging arithmetic addition trees.

The circuits in Table VI compute multiply/add expressions. Circuit MAC1 computes the expression $\mathtt{y} = \mathtt{a0} * \mathtt{b0} + \mathtt{a1} * \mathtt{b1}$, circuit MAC8 computes $\mathtt{y} = \mathtt{a0} * \mathtt{b0} + \cdots + \mathtt{a8} * \mathtt{b8}$. In the implementation circuit, the multiplication operations cannot be identified as individual blocks. The arithmetic optimization by the synthesis tool has merged all addition circuits of the "$*$" and "$+$" operators and optimized the resulting addition network. Although the individual operators can no longer be identified, the overall structure of the circuit is composed of a stage computing partial products and an addition circuit, similar to the structures shown in Figs. 2 and 3. Hence, our method can be applied in the usual way. We generated a specification for the circuit as it would be produced by the front-end of an equivalence checker. Then, the extraction procedure was run on the implementation. It generated an arithmetic bit level representation of the merged addition circuit where the partial product bits of the specification serve as primary addends to the half adder network. Just like for the multipliers of Tables IV and V the correctness of the computed expressions was verified using Lemma 2.

Column 2 of Table VI shows the number of multipliers in each expression. The remaining columns report the bit width of the result $y$, the size of each circuit given as the number of connections, and the CPU time needed to extract the arithmetic bit-level representation.

## VI. CONCLUSION

In this paper, we propose a method for equivalence checking of arithmetic circuits including multipliers. The method is based on a bit-level reverse-engineering approach. The main challenge is to efficiently extract an arithmetic bit level description of a circuit from a given gate netlist. The presented extraction algorithms have been tested on various multipliers and other arithmetic circuits and proved very promising. The approach effectively complements conventional equivalence checking frameworks and can increase the robustness of equivalence checking for arithmetic circuits.

## ACKNOWLEDGMENT

The authors are grateful to S. Höreth (Infineon Technologies, Munich, Germany) and T. Rudlof (formerly with SIEMENS AG, Munich, currently with Mentor Graphics, Inc., Billerica, MA) for fruitful discussions and for providing the multiplier examples generated by commercial synthesis tools.

## REFERENCES

[1] D. Brand, "Verification of large synthesized designs," in *Proc. Int. Conf. Computer-Aided Design*, 1993, pp. 534–537.
[2] W. Kunz, "An efficient tool for logic verification based on recursive learning," in *Proc. Int. Conf Computer-Aided Design*, Nov. 1993, pp. 538–543.
[3] A. Kühlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Proc. Design Automation Conf.*, June 1997, pp. 263–268.
[4] J. R. Bitner, J. Jain, M. S. Abadir, J. A. Abraham, and D. S. Fussell, "Efficient algorithmic circuit verification using indexed BDDs," in *Proc. Fault Tolerant Comput. Symp.*, 1994, pp. 266–275.
[5] J. Jain, R. Mukherjee, and M. Fujita, "Advanced verification techniques based on learning," in *Proc. 32nd ACM/IEEE Design Automation Conf.*, June 1995, pp. 420–426.
[6] Y. Matsunaga, "An efficient equivalence checker for combinational circuits," in *Proc. Design Automation Conf.*, June 1996, pp. 629–634.
[7] R. Bryant and Y. A. Chen, "Verification of arithmetic circuits with binary moment diagrams," in *Proc. Design Automation Conf.*, 1995, pp. 535–541.
[8] T. Stanion, "Implicit verification of structurally dissimilar arithmetic circuits," in *Proc. Int. Conf. Comput. Design.*, Oct. 1999, pp. 46–50.
[9] M. Fujita, "Verification of arithmetic circuits by comparing two similar circuits," in *Proc. Int. Conf. Computer-Aided Verification.*, R. Alur and T. A. Henzinger, Eds, Aug. 1996, pp. 159–168.
[10] Y.-T. Chang and K.-T. Cheng, "Induction-based gate-level verification of multipliers," in *Proc. Int. Conf. Computer-Aided Design.*, San Jose, CA, 2001, pp. 190–193.
[11] ——, "Self-referential verification of gate-level implementations of arithmetic circuits," in *Proc. Design Automation Conf.*, 2002, pp. 311–316.

[12] W. Kunz and D. Stoffel, *Reasoning in Boolean Networks—Logic Synthesis and Verification Using Testing Techniques*, MA: Kluwer, 1997.
[13] H. Simonis, "Formal verification of multipliers," in *Proceedings of the IFIP WG10.2 WG10.S International Workshop on Applied Formal Methods for Correct VLSI Design*, L. J. Claesen, Ed.. North Holland, The Netherlands, 1990, pp. 267–286.
[14] Y.-A. Chen and J.-C. Chen, "Equivalence checking of integer multipliers," in *Proc. Asia South Pacific Design Automation Conf.*, Yokohama, Japan, 2001, pp. 196–174.
[15] K. Hamaguchi, A. Morita, and S. Yajima, "Efficient construction of binary moment diagrams for verifying arithmetic circuits," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1995, pp. 78–82.
[16] A. Kuehlmann, M. K. Ganai, and V. Paruthi, "Circuit-based Boolean reasoning," in *Proc. Design Automation Conf.*, June 2001, pp. 232–237.
[17] B. Parhami, *Computer Arithmetic—Algorithms and Hardware Designs*. New York: Oxford Univ. Press, 1999.
[18] T. Larrabee, "Efficient generation of test patterns using boolean difference," in *Proc. Int. Test Conf.*, 1989, pp. 795–801.
[19] M. Keim, M. Martin, B. Becker, R. Drechsler, and P. Molitor, "Polynomial formal verification of multipliers," in *Proc. VLSI Test Symp.*, 1997, pp. 150–155.
[20] R. P. Brent and H. T. Rung, "A regular layout for parallel adders," *IEEE Trans. Comput.*, vol. C-31, pp. 260–264, Mar. 1982.

**Dominik Stoffel** (M'95) obtained the Diplom-Ingenieur degree in electrical engineering from the University of Karlsruhe, Karlsruhe, Germany, in 1992 and the Ph.D. degree in computer science from the University of Frankfurt, Frankfurt, Germany, in 1999.

From 1994 to 1998, he was with the Max-Planck Fault-Tolerant Computing Group in Potsdam. From 1998 to 2001, he was with the Electronic Design Automation Group, University of Frankfurt. Since 2001, he has been a Postdoctoral Researcher in the Electronic Design Automation Group, University of Kaiserslautern, Kaiserslautern, Germany. His research interests are in the field of formal hardware verification and logic synthesis.

**Wolfgang Kunz** (S'90–M'91) obtained the Dipl.Ing. degree of electrical engineering from University of Karlsruhe, Karlsruhe, Germany, in 1989 and the Ph.D. degree from the University of Hannover, Hannover, Germany, in 1992.

From 1993 to 1998, he was with Max Planck Fault-Tolerant Computing Group, University of Potsdam, Potsdam, Germany. From 1998 to 2001, he was a Professor in the Computer Science Department, University of Frankfurt, Frankfurt, Germany. Since 2001, he has been with the Electrical Engineering Department, University of Kaiserslautern, Kaiserslautern, Germany. He conducts research in the areas of logic and layout synthesis, equivalence checking, and ATPG.

Prof Kunz has received several awards including the IEEE Transactions on Computer-Aided Design Best Paper Award.