Verification of Integer Multipliers on the Arithmetic Bit Level

Dominik Stoffel Wolfgang Kunz
Institute of Computer Science, University of Frankfurt/Main, Germany

Abstract

One of the most severe short-comings of currently available equivalence checkers is their inability to verify integer multipliers. In this paper, we present a bit level reverse-engineering technique that can be integrated into standard equivalence checking flows. 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. Experimental results show the promise of our approach.

1 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* [2, 10], also called internal *cut points* [9]. Techniques to exploit these similarities have enabled equivalence checkers to verify very large combinational circuits as has been shown by several authors [1, 2, 7, 9, 10, 12]. 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 an RT-level (*register transfer 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 is bound to fail. The problem is that the gate-level model generated by the equivalence checker will look entirely different compared to 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 BMDs [3] 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 [1, 13] 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” [13]) are exploited.

An approach based on a standard equivalence checking engine was proposed by Fujita [5]. 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 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.

Reverse engineering could be considered as a very 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 frontend 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 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 look at the following four ways of multiplying two decimal numbers.

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

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 by means of 16x16 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 [11], 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. However, 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.

In general terms, the proposed approach can be summarized as follows:

1. Decompose the two combinational circuits – where possible – into networks of 1-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.

2 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 two's complement notation from a number Y, for example, is implemented by inverting all bits of X, adding 1, and adding Y. Also multiplication is based on addition. Hardware multipliers most often are composed of two stages (Fig. 1). In the first stage, the partial products are generated from the two operand vectors, X and Y. The way these 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 the inputs to the second stage, which is an addition circuit. We will call the inputs to an addition circuit *primary addends*, in the sequel. The addition circuit adds up the primary addends 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 *carry-save adders (CSA)* or a *Wallace tree*.

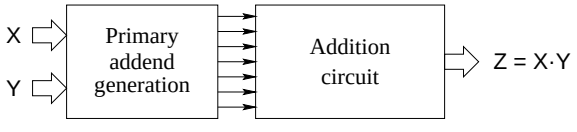

Figure 1: Basic multiplier structure

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. A half adder is a circuit that arithmetically adds two binary operands and produces two binary results, a sum and a carry signal. Figure 2 shows the gate schematics of a half adder. In the sequel, we will use the half adder symbol shown on the right side of Figure 2.

Note that a full adder can be assembled from *three* half adders. Figure 3 shows a possible implementation of a full adder and the

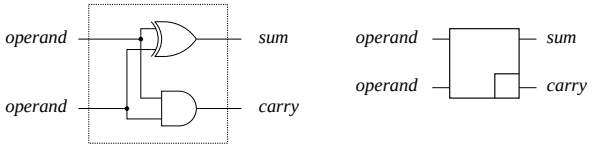

Figure 2: Half adder, schematics and symbol

corresponding half adder network. The third half adder, R, adds the two carry bits c_1 and c_2 of the other half adders, P and Q, and produces the full adder carry output w. Because the two signals c_1 and c_2 can never assume the logic value 1 at the same time, the carry output of the third half adder produces a constant 0.

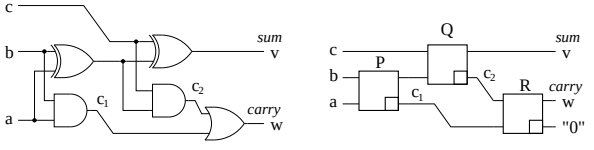

Figure 3: Full adder decomposed into half adders

Once we have 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. We now 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 bipartite directed graph with vertex set* V *and directed 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.

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

The addition graph is associated with a half adder network as follows. Each sum node is associated with the sum output of a half adder in the network. Each carry node is associated with the carry output of a half adder. Each primary addend 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, Figure 4 shows the addition graph of the full adder of Figure 3. Note that the primary addends and the carry nodes are the source nodes of an addition graph, and

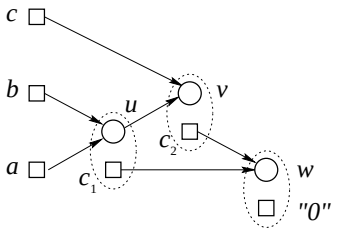

Figure 4: Addition graph for full adder

are also referred to as *addends* in the following. In Figure 4, 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.

The modelling 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.

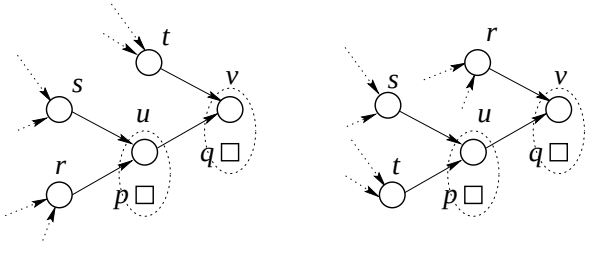

Figure 5: Addition graph of Lemma 1

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 Figure 5. 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 $\tilde{T}$ (Figure 6) 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. □

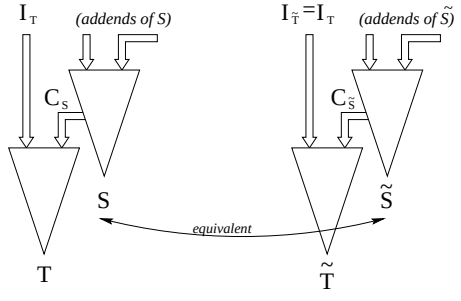

Figure 6: Illustration of Lemma 2

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 $\tilde{S}$ 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 run-time 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 subject of the following section.

3 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 4-bit carry-lookahead adder of Figure 7. In order to speed up computation time, the carry signals in each output cone are generated by a special logic block.

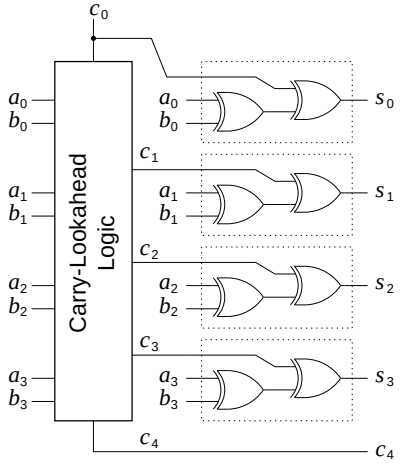

Figure 7: 4-bit carry-lookahead adder

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.

3.1 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 [11] to detect XOR relationships in the original circuit. 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 shown in Figure 8.

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

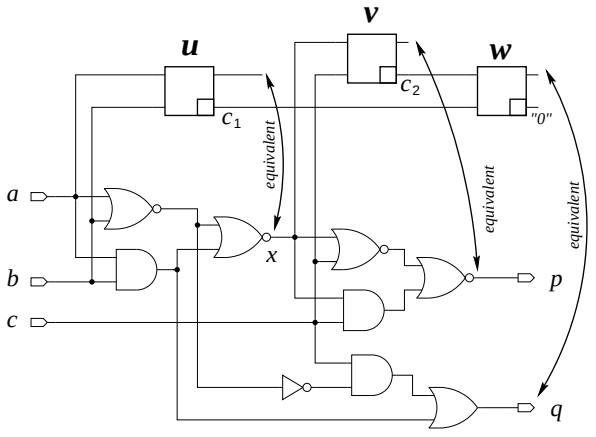

Figure 8: Full adder implementation and mapped half adder network

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 ($y = a \oplus b = \overline{a} \oplus \overline{b}$), 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 Boolean analysis underlying this procedure is local and of fairly low complexity. An efficient implementation can be based not only on implication techniques but just as well on decision diagrams, SAT solving or structural hashing [9].

3.2 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 3.1 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 retry to map the unmapped nodes using the new half adders as operands.

As an example, consider a circuit computing some additions ac-

cording to the half adder network shown in Figure 9. In this network,

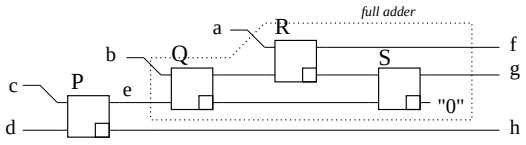

Figure 9: Arithmetic bit level representation (example)

function f is the output of a chain of half adders. Signal d traverses three XOR stages before reaching f. In a practical implementation, it may be of advantage to compute f by an XOR tree rather than a chain. Figure 10 shows such an implementation. Also shown are the half adders inserted after applying the basic procedure described in Section 3.1. Note that in this example it is not possible to express the

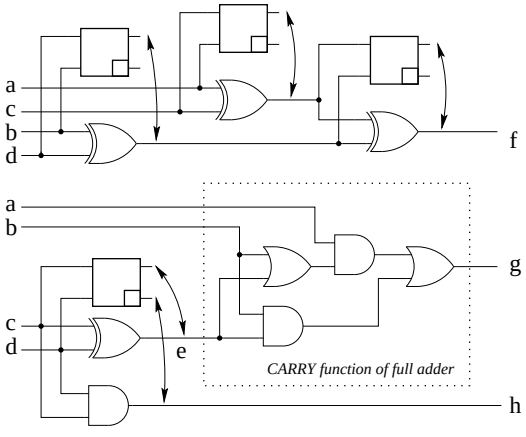

Figure 10: Implementation of example of Fig. 9 with added reference circuitry

carry function of signal g as an XOR of any two half adder signals in the reference circuit. Hence, we fail to completely map the gate netlist to a half adder network as in Figure 9.

If, as in this example, the computation of sum and carry signals has been locally separated and the XOR trees in the sum have been restructured, certain carry functions can no longer be expressed in terms of the available signals in the reference circuit and cannot be mapped. This leads to "gaps" in the extracted network.

In order to map such gaps, we proceed as follows. First, for each gap, we identify its mapped inputs. Then, by a topological analysis in the extracted half adder network, we identify the signal where the sum of the input functions is computed. We then restructure the half adder network using commutative and associative laws such that the operands needed to map the gap functions are produced. For example, in Figure 10, signals a, b and e are the inputs of a gap. We search in the half adder network a signal computing the sum of a, b and e. By backtracing in the addition graph, we determine for each input its addends (primary addends and carry nodes). Then, by forward tracing we identify the sum node summing up these addends. In Figure 10, we identify function f which computes the sum of a, b, c and d. Next, we restructure the half adder network such that an addition chain with the operands a, b and e is obtained and locally extend the half adder network by this addition chain. If several addition chains are possible, all of them are inserted. In our example, the addition chain is the series of half adders P, Q and R of Figure 9 and can be added (not shown) to the half adder network of Figure 10. Now, signal g can be expressed as the XOR of two carry signals in the reference circuit, yielding the half adder S of Figure 9 and completing the mapping.

3.3 Algorithm

```
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 nodes y in C {
    locally extend half adder network R for 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;
}
```

Table 1: Algorithm for half adder network extraction

Table 1 shows the pseudo-code 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 Figure 8. The third phase targets the remaining unmapped nodes as described in Section 3.2. In each of these phases, subroutine *extraction_pass()* shown in Table 2 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

```
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 2: Subroutine performing a half adder extraction pass

equivalence check of Section 2.

Note that our procedure is robust also in cases where the basic building blocks are not half or full adders. Consider the example in Figure 7. In the first phase of the algorithm of Table 1 we identify the XORs performing the additions. For each XOR a half adder is inserted in the reference circuit. In the second phase we express each of the outputs c_1, c_2, c_3 and c_4 of the carry-lookahead logic as XORs in terms of carry outputs of the inserted half adders, completing the mapping. It is interesting to note that the resulting half adder network is of carry-propagate ("ripple carry") structure.

4 Verification Framework

The proposed approach can be added as an additional heuristic to existing equivalence checking frameworks. Equivalence checking is run for given circuits in the usual way until standard techniques abort by lack of internal equivalences. If there are large regions without internal equivalences, the extraction procedure of Section 3 is activated, attempting to generate an arithmetic bit level representation of the pathological region. This can be successful, if the region is indeed an arithmetic block. If the circuit contains a multiplier, standard equivalence checking will be successful in identifying internal equivalences for many nodes in the circuit, including the partial products of the multiplier. However, it will fail to process the subsequent addition circuit. After extracting the arithmetic bit level representation the verification can be completed.

In this paper, we focus on verifying the equivalence of addition circuits with dissimilar structure as they appear in different multiplier architectures. Another multiplier architecture parameter is the use of Booth recoding, which affects not the addition circuit but the primary addend generation step of Figure 1. The multiplicand is re-encoded to produce a smaller set of partial products to be accumulated by the addition circuit. In order to verify multipliers with Booth recoding in a verification framework using the proposed approach, it is necessary that the frontend producing the gate-level description of the specification generates both, the non-Booth-encoded and the Booth-encoded partial products bits. The equivalence checker will then express the extracted half adders in whatever partial products have been used in the design under verification. We have not yet implemented this in our verification tool, therefore the experimental results of Section 5 have been obtained for non-Booth-encoded multipliers only.

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. Observe that multipliers are highly random-pattern testable so that a buggy design is usually detected by only a small number of random patterns.

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 [3, 4, 8] that knowledge about the subcomponents of a multiplier can be very useful in BMD construction. It seems likely that the arithmetic bit level representation as extracted by the procedure of Section 3 could be a good basis for heuristically guiding a BMD construction process along the lines of [4, 6].

5 Experimental Results

The described techniques have been implemented as a part of the HANNIBAL [11] tool. Table 3 shows the results for extracting the half adder networks for a number of multiplier circuits. The first column shows the circuit name, the next three columns show the bit widths of multiplication operands, X, Y, and result, Z, and the last column shows the run time of the algorithm. The CPU times are given in seconds on a 450 MHz PC running Linux.

circuit name	bit vector widths X	Y	Z	CPU time (secs.)
mult8x8	8	8	16	3
dw_csa_8x8	8	8	16	3
dw_nbw_8x8	8	8	16	12
mult16x16	16	16	32	40
dw_csa_16x16	16	16	32	34
dw_nbw_16x16	16	16	32	132
c6288	16	16	32	76
c6288nr	16	16	32	56
c6288opt	16	16	32	36
dw_csa_16x26	16	26	42	98
dw_nbw_16x26	16	26	42	156

Table 3: Experimental results for half adder extraction

Circuits *mult8x8* and *mult16x16* are 8- and 16-bit multipliers produced by a self-written generator for multipliers in CSA array architecture. The circuits denoted by prefix *dw_csa* and *dw_nbw* are multipliers in CSA array and Wallace tree architecture, respectively. They have been created using a commercial CAD system (Synopsys Design Compiler). Circuit *c6288* is the well-known 16x16 bit multiplier from the ISCAS-85 benchmark set, circuit *c6288nr* is its non-redundant version, and circuit *c6288opt* is the result of optimizing *c6288* using SIS with *script.rugged*.

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.

We verified the equivalence between any pair of multipliers with the same operand widths 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.

6 Conclusion

In this paper, we propose a method for equivalence checking of integer multipliers 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 different multiplier architectures and proved very promising. We are currently extending our tool to different types of primary addends so that Booth-recoded multipliers can also be handled. The presented approach can easily be integrated into standard equivalence checking frameworks and can increase the robustness of conventional equivalence checkers for arithmetic circuits.

7 Acknowledgment

We are grateful to Stefan Höreth and Thomas Rudlof from SIEMENS, ZT SE 4, for fruitful discussions and for providing the multiplier examples generated by a commercial synthesis tool.

References

[1] 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 Computing Symposium (FTCS-94)*, pp. 266–275, 1994.

[2] D. Brand, "Verification of Large Synthesized Designs," in *Proc. Intl. Conf. on Computer-Aided Design (ICCAD-93)*, pp. 534–537, 1993.

[3] R. Bryant and Y. A. Chen, "Verification of Arithmetic Functions by Binary Moment Diagrams," in *Proc. Design Automation Conference (DAC-95)*, pp. 535–541, 1995.

[4] Y.-A. Chen and J.-C. Chen, "Equivalence Checking of Integer Multipliers," in *Proc. Asia and South Pacific Design Automation Conference (ASPDAC-01)*, (Yokohama, Japan), 2001.

[5] M. Fujita, "Verification of Arithmetic Circuits by Comparing Two Similar Circuits," in *Proc. International Conference on Computer Aided Verification (CAV '96)*.

[6] K. Hamaguchi, A. Morita, and S. Yajima, "Efficient Construction of Binary Moment Diagrams for Verifying Arithmetic Circuits," in *Proc. Internation Conference on Computer-Aided Design (ICCAD-95)*, pp. 78–82, November 1995.

[7] J. Jain, R. Mukherjee, and M. Fujita, "Advanced Verification Techniques Based on Learning," in *Proc. 32nd ACM/IEEE Design Automation Conference (DAC-95)*, pp. 420–426, June 1995.

[8] M. Keim, M. Martin, B. Becker, R. Drechsler, and P. Molitor, "Polynomial Formal Verification of Multipliers," in *VLSI Test Symp.*, pp. 150–155, 1997.

[9] A. Kühlmann and F. Krohm, "Equivalence Checking Using Cuts and Heaps," in *Proc. Design Automation Conference (DAC-97)*, pp. 263–268, Nov. 1997.

[10] W. Kunz, "An Efficient Tool for Logic Verification Based on Recursive Learning," in *Proc. Intl. Conference on Computer-Aided Design (ICCAD-93)*, pp. 538–543, Nov. 1993.

[11] W. Kunz and D. Stoffel, *Reasoning in Boolean Networks - Logic Synthesis and Verification Using Testing Techniques*. Boston: Kluwer Academic Publishers, 1997.

[12] Y. Matsunaga, "An Efficient Equivalence Checker for Combinational Circuits," in *Proc. Design Automation Conference (DAC-96)*, pp. 629–634, June 1996.

[13] T. Stanion, "Implicit Verification of Structurally Dissimilar Arithmetic Circuits," in *Proc. International Conference on Computer Design (ICCD-99)*, pp. 46–50, October 1999.