Verifying Full-Custom Multipliers by Boolean Equivalence Checking and an Arithmetic Bit Level Proof

Udo Krautz, Markus Wedler, Wolfgang Kunz, Kai Weber, Christian Jacobi, Matthias Pflanz

***Abstract*—In this paper we describe a practical methodology to formally verify highly optimized, industrial multipliers. We define a multiplier description language which abstracts from low-level optimizations and which can model a wide range of common implementations at a structural and arithmetic level. The correctness of the created model is established by bit level transformations matching the model against a standard multiplication specification. The model is also translated into a gate netlist to be compared with the full-custom implementation of the multiplier by standard equivalence checking. The advantage of this approach is that we use a high level language to provide the correlation between structure and bit level arithmetic. This compares favorably with other approaches that have to spend considerable effort on extracting this information from highly optimized implementations. Our approach is easily portable and proved applicable to a wide variety of state-of-the-art industrial designs.**

***Index Terms*—formal verification, algorithm**

I. Introduction

FORMAL property checking has gained significant importance in System-on-Chip (SoC) verification and has become part of many industrial design flows.Unfortunately, arithmetic circuits with multiplication have always been - and to some extent still are - the show stopper for formal property checking in industrial practice. Neither satisfiability (SAT) solving nor decision diagrams of any sort provide robust and universal frameworks to deal with arithmetics. Specialized "engineering" solutions are available that adapt to specific scenarios in equivalence checking or property checking. Most of these methods depend on exploiting specific high-level arithmetic information. This can be useful for highly regular designs as they may result from automatic module generation. However, for full-custom logic design the problem, so far, has remained unsolved.

When designing arithmetic units for high-performance applications a designer will usually start implementing a basic version of the algorithm. At this point, word-level abstractions are still available in the design. However, as aggressive timing requirements have to be met, the initial algorithm will be be modified and optimized by a series of manual steps involving transformations at all levels. Such a full-custom implementation is not only of high complexity, its specialized structure makes it difficult to apply any kind of abstraction above the Boolean bit-level. Obviously, designs resulting from such a manual optimization process may contain hard to find errors that will surface late in the design cycle and may not be found by simulation or emulation. The famous Intel Pentium division bug [8] resulted from such circumstances. Even after many years, and in spite of substantial progress in formal verification, functional correctness of full-custom arithmetics has remained a major concern in SoC design flows.

Multipliers have become very common in today's designs of processors, digital signal processors, hardware accelerators and other signal processing devices. While standard property checking usually fails for such designs, in industrial practice it is often attempted to verify these designs by equivalence checking against some reference. This can work well if reference and design have a large amount of structural similarities and share many functionally equivalent signals. But if design and reference have different architectures the equivalence check immediately becomes impossible.

This paper presents a new and convenient methodology for proving the correctness of multiplier and multiply-accumulate circuit designs in a full custom design flow. It utilizes a basic description of the implemented algorithm which is created in early phases of the design flow and requires only little extra work for the designer who spends most of the time in full-custom optimizations. This specification defines the arithmetic circuit at the arithmetic bit level and allows for generation of a gate level netlist. In this way, a large amount of structural similarity with the design is obtained so that a standard equivalence checker can be utilized to verify the design against the specification. Furthermore, the correctness of the specification is proven by arithmetic bit-level reasoning.

A typical application of our method can be found in the context of verifying floating-point-units (FPUs). Their embedded multipliers are separated for formal approaches, e.g. by black boxing [2], and often checked by simulation. With the proposed technique, the designer will provide a high level description of the implemented algorithm that allows early verification of the multiplier's datapath.

The paper is organized as follows. At first, we give an overview of previous approaches to verify multipliers. We continue by introducing our methodology, we provide basic definitions and illustrate the individual steps of our proof. We further present extensions to our approach, that enable us to solve more complex arithmetic problems. The paper concludes with a presentation of experimental results on industrial full-custom multipliers.

II. Previous Work

Multipliers lack a compact canonical representation that can be built efficiently from gate level implementations. For ROBDDs [1] it is well-known that the number of nodes grows exponentially with the number of input bits to the multiplier[1]. Even if BDDs are not directly used to build the multiplier outputs but only certain internal relations, like in the implicit approach proposed by Stanion[10], they lack robustness and suffer from BDD node explosion. Lamb [7] showed an optimal partitioning of a multiplier so that the BDD

Udo Krautz, Markus Wedler, Wolfgang Kunz are with the University of Kaiserslautern {krautz | wedler | kunz@eit.uni-kl.de} (P.O.Box 3049; 67653 Kaiserslautern; Germany)

Kai Weber, Christian Jacobi, Matthias Pflanz are with IBM Deutschland Entwicklungs GmbH {kai.weber | cjacobi | mpflanz@de.ibm.com}(Schoenaicher Str. 220; 71032 Boeblingen; Germany)

978-1-4244-1922-7/08/$25.00 ©2008 IEEE

does not grow faster than the square of the number of inputs, however it is not clear whether such a partitioning always exists and how it can be found for arbitrary designs.

There have been several attempts to solve the multiplier verification problem with different variants of decision diagrams. Bryant and Cheng used word level decision diagrams called binary moment diagrams (BMDs) [4] to efficiently represent integer multiplication. In a *BMD the number of nodes to represent integer multiplication only grows linearly in contrast to the exponential growth of a BDD. The authors also gave a hierarchical verification approach, which, however, requires a manual partitioning. Hamaguchi et. al. showed an efficient backward construction for *BMDs [11] to overcome this manual partitioning, but Wefel and Molitor showed cases where constructing the *BMD for a faulty multiplier in this way suffers from exponential growth of the diagram [9]. Several improvements to Hamaguchi's method have been made [12] but the main obstacle remains - BMDs require word level information about the design which usually is not available or is very hard to extract from highly optimized gate level descriptions being typical in high performance applications. When building *BMDs for such highly irregular multipliers the same node explosion can be observed during the construction process as for BDDs, in spite of the theoretical result that the final representation will be compact.

Several approaches propose functional decomposition of multipliers [3], [13], [14] in order to prove subproblems and solve global equivalence by induction or interpolation. These again require manual partitioning of partial products and intermediate sums that may be hard to find when complex implementations are under examination.

In [16], [17] a combination of theorem proving and symbolic trajectory evaluation [18] is used to verify multiplier designs. Both approaches decompose the general proof into several proof steps. A theorem prover is used to show the validity of the decomposition. At the lowest proof level, properties are given in form of pre- and postconditions and checked by symbolic trajectory evaluation. Since some preconditions have to be defined on intermediate results of the design, the lower level proofs may still require considerable manual effort to be carried out.

In [5] an equivalent network of bit adders is extracted from a gate level description of arithmetic circuits. Equivalence between two extracted networks is proven by a simple calculus. In [15] this work is extended towards applications in property checking. They provide a normalization process that creates structural similarities between the design under verification and the specification, given as a property. However, it is assumed that not only the property but also the design is specified at the arithmetic bit level or higher levels of abstraction.

Both approaches ([5], [15]) rely on the successful extraction of the arithmetic bit level information. This is possible for synthesized netlists but in full-custom design the situation is different. Manual architectural changes and full-custom optimizations may involve global transformations. Extraction of an arithmetic bit-level description can turn out to be quite difficult since there is an exponential number of possible decompositions.

III. Methodology

A. Definitions

In this section we will outline our methodology for multiplier verification. Throughout this paper the term *multiplication* refers to *integer multiplication*. After some basic definitions that are summarized in subsection III-A we will start with an overview of the proposed methodology in subsection III-B. The details of this methodology are then elaborated in subsections III-C, III-D and III-E.

In the remainder of the paper we will use the following notations:

- a is a vector of n Boolean variables $a = (a_{n-1}, \ldots, a_0) = a[n-1:0]$
- The natural number represented by a is $\langle a \rangle = \sum_{i=0}^{n-1} a_i \cdot 2^i$.
- With $a = a[n-1:0]$ and $b = b[m-1:0]$ the product of both shall be defined as $\langle r \rangle = \langle a \rangle \cdot \langle b \rangle$ and $r = r[m+n-1:0]$.

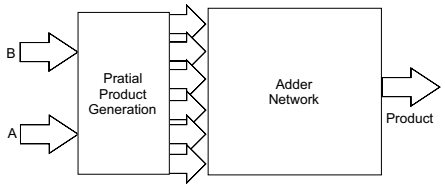

Fig. 1. Basic multiplier architecture

Typically a multiplier can be divided into two portions (Figure 1) a generator of "partial products" and the adder-tree to generate the multiplication result. We first define generation of partial products:

Definition 1: Partial product generation calculates a set of vectors $pp_j = pp_j[m+n-1:0]$ for vectors $a = a[n-1:0]$ and $b = b[m-1:0]$. Usually they are defined by bitwise multiplication $pp_j = a \cdot b[j] \cdot 2^j$ or some Booth-encoding $pp_j = a \cdot B_j$. It holds:

$$\langle r \rangle = \sum_j \langle pp_j \rangle = \sum_{k=0}^{m+n-1} r_k \cdot 2^k.$$

Notice that partial products may contain p leading zeros and q trailing zeros $pp_j = (0^p, pp_{j,m+n-1}, \cdots, pp_{j,0}, 0^q)$. The second portion is a network of adders, which is used to sum up the partial products to the final multiplication result.

Definition 2: Following [15], an addition network is a set of Boolean variables A, being the addends to the network. An addend $a \in A_i$ contributes to column i of the network. Addends are weighted by signed integers $\omega_i : A_i \rightarrow \mathbb{Z}$. The result of the addition network $r = r[m+n-1:0]$ may be defined as the sum of weighted addition network outputs.

$$\langle r \rangle = \sum_{i=0}^{m+n-1} 2^i \cdot \Big(\sum_{a \in A_i} \omega_i(a_i) \cdot a_i \Big)$$

B. Overview

The basic flow for multiplier verification used in our approach is depicted in Figure 2. In addition to the design under verification we provide an abstract reference at the word level. The language used to define this reference will be outlined in Subsection III-D. We use this language to formulate a detailed specification of the algorithm implemented by the design under verification. This description is similar to initial implementations that a designer will create when starting to implement a given multiplier algorithm.

In contrast to references specified in standard property languages, we provide a mechanism to translate the reference into a gate netlist that shows significant structural similarity with the design under verification. Hence, a standard equivalence checker can be used to prove equivalence between reference and implementation. Furthermore, our specification language provides arithmetic functions that we use to specify partial products and the addition network of our implementation. Consequently, it is very easy to extract the adder network. We prove that the reference is a correct model for multiplication with a series of simple transformations of the network that lead to a standard representation for multiplication. We prove that the reference is a correct model for multiplication with a simple transformation algorithm that produces a standard representation for multiplication. The transformations will be described in III-C. The transformation process is fully automated.

Correctness of the design will be concluded from arithmetic correctness of the reference in conjunction with gate level equivalence between reference and design.

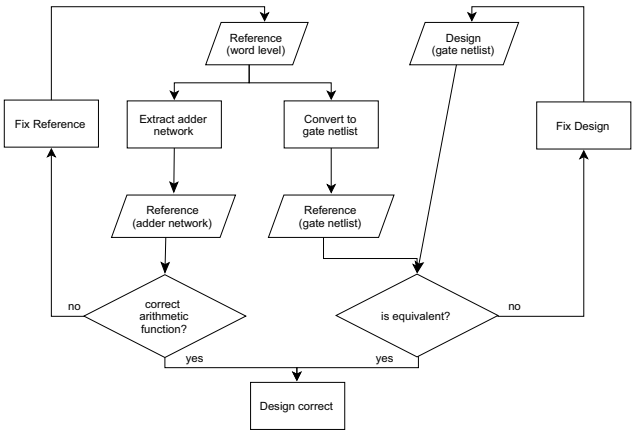

Fig. 2. Methodology overview on multiplier verification

In some implementations multipliers are reused to implement more than one function. For example, several integer multiply-add operations can be calculated in parallel given a multiplier of sufficient bit width. The underlying arithmetic algorithms for such functions are slightly different from a standard multiplication algorithm. In our work, this is approached by creating an individual specification and performing a separate verification run for each function.

C. Proving arithmetic correctness

In the following, we will describe our transformation algorithm to prove correctness of a multiplication algorithm. The transformations utilize associative and commutative laws for bit level addition. Note, that in contrast to [15] we do not need the distributive laws.

Using these laws we can verify implementations with various optimizations, e.g., Booth-encoding. This is achieved by expressing every ARDL function at the arithmetic bit level, thereby constructing an adder network of the reference. Through a series of transformations on this network we can restore the basic definition of multiplication at the bit level:

$$\langle a\rangle \cdot \langle b\rangle = (\sum_i a_i \cdot 2^i) \cdot (\sum_j b_j \cdot 2^j) = \sum_k 2^k \sum_{k=i+j} a_i \cdot b_j$$

This equation defines an addition network where the $(a_i \cdot b_j)$ are addends to column $k = i + j$ with weight $\omega(a_i b_j) = 1$:

$$\langle r\rangle = \sum_k 2^k \cdot \sum_{k=i+j} a_i \cdot b_j \cdot \omega_{i,j}(a_i b_j).$$

This corresponds to multiplication learned in grade-school which is often expressed in matrix notation as follows:

$$P = \begin{array}{ccccc} \cdots & \cdots & a_2b_0 & a_1b_0 & a_0b_0 \\ \cdots & a_2b_1 & a_1b_1 & a_0b_1 & 0 \\ a_2b_2 & a_1b_2 & a_0b_2 & 0 & 0 \end{array}$$

In the following, we will investigate how to transform other multiplication schemes used for implementation into this basic reference scheme.

Many implementation techniques for multipliers focus on reducing the number of partial products. This will reduce the number of additions necessary to calculate the final result and will therefore reduce delay and area of the implementation. A prominent technique for this purpose is called Booth encoding. A prominent technique for multiplier optimization is reducing the number of partial products by Booth encoding. The standard scheme for multiplication defines partial products by $pp_j = a \cdot b[j] \cdot 2^j$. These products correspond to the lines given in matrix-notation of P. Booth encodings define a *Booth-digit* B_j that subsumes several consecutive bits of b. The precise definition of the *Booth-digit* B_j depends on the chosen radix. The prevalent version used in industry is radix-4/ Booth-2 encoding, where the *Booth-digit* is defined as $B_{2j} = -2b[j+1]+b[j]+b[j-1]$. In this version of Booth-2 encoding, the partial products are defined as $pp_j = a \cdot B_{2j}$. Obviously, pp_j subsumes several partial products of the basic multiplier definition. In order to facilitate a transformation of the addition scheme generated by the algorithm with Booth-encoding into the basic multiplication scheme we have to consider independent partial products rather than their aggregation. Hence, we implement the partial products of our reference by the expanded definition $pp_j = pp_{1,j} + pp_{2,j} + pp_{3,j}$ where the appropriate three partial products $pp_{1,j} = -2b[j+1] \cdot a \cdot 4^j$, $pp_{2,j} = b[j] \cdot a \cdot 4^j$ and $pp_{3,j} = b[j-1] \cdot a \cdot 4^j$ are added.

Booth encoding therefore results in negative weights $\omega_{ij}(a_i b_{j+1}) < 0$ for $pp_{1,j}$. At the bit level these negative numbers will be represented in two's complement. This would

result in negated addends in the addition network. However, these negated addends will be canceled out throughout the addition network. In implementations all negative addends are created by inversion and addition of an additional *hot-one* bit to an appropriate column. Inversion of a single bit a_i can be represented by $1 - a_i = \overline{a_i}$. When an inverted vector $\overline{a}$ is added to the addition network, we replace every addend $\overline{a_i}$ to column i by two addends $-a_i$ and an additional addend $c_i = 1$. These constants will cancel out with the hot-one bit specified in the reference model. After these transformations the resulting addition network significantly differs from the basic multiplier scheme. The observed differences can be summarized as follows:

- negative weights in columns due to subtraction
- shifted addends
- addends are added into multiple columns

However, if the underlying multiplication algorithm is sound all these differences will cancel out each other. For instance, addends that seem to be added twice into the network will be subtracted again later.

To transform arbitrary multiplication into the standard scheme we use the following transformations in our algorithm.

1. Normalization: Let M be an addition network with addends $a_i b_j$. In the normalized addition network $N(M)$ all occurrences of an addend $a_i b_j$ are moved to column $k = i + j$ and the weight of $\omega_k(a_i b_j)$ is modified accordingly.

2. Purging: In each column k of an addition network M multiple instances $pp_{l-1} \cdots pp_0$ of an addend $a_i b_j$ are summarized by addition of the weights $\omega_k(a_i b_j) = \sum_{s=0}^{l-1} \omega_k(pp_s)$.

Both transformations lead to an equivalent addition network where the order of addition has been rearranged at bit level and optimizations of the implementation have been reversed. It is possible to transform any equivalent addition network M to the basic multiplication scheme by $M' = P(N(M))$. In the resulting addition network M' we only have to check whether $\omega_{i+j}(a_i b_j) = 1$ and $\omega_k(a_i b_j) = 0$ is valid for all i, j and $k \neq i + j$. In this case we have proven correctness of the initial addition network and hence of the underlying algorithm. If differences remain through the transformation, these will be reported as arithmetic errors in the reference. The runtime of proofs on erroneous references does not exceed that of correct references.

D. Reference Description Language

In this section we describe the proposed arithmetic reference description language (ARDL) used to model the structure for implementations of multiplier-like functions. ARDL is used to create specifications on an abstraction layer between the gate level description of the design and the word level description of the corresponding arithmetic function. ARDL is restricted to combinational models only. Sequential implementations can be handled by unfolding into a combinational netlist.

According to the flow of Figure 2 the model described in the proposed language is used to generate a gate level as well as an arithmetic bit level description of the reference. In this way, our language facilitates arithmetic reasoning at the bit level, in combination with a structural view of the implementation. Its syntax is close to the usual HD languages like VHDL or Verilog but only contains a few combinational functions to describe basic multiplier elements.

Developing a reference model is quite natural in a typical design process. Often a very similar but informal specification is created by designers anyway when starting to develop a new multiplier. Our language could substitute such an informal description.

In the first section, named *variables,* of every ARDL reference description the variables used for inputs, outputs and intermediate results are declared. This section, name variables, can be easily translated into the corresponding VHDL declarations. We use the following notations:

- $a = (a_l, ..., a_0)$ always denotes a vector of Boolean variables called *bit vector*
- I, O are sets of *bit vectors*

We call every $a \in I$ an input and $a \in O$ an output. ARDL requires that $I \cap O = \{\varnothing\}$. The keywords *in* and *out* indicate whether a declared variable is an input or an output. We also declare variables used as partial products with the keyword *pp*. Remaining intermediate variables of the adder tree are automatically determined.

The following two sections of an ARDL specification are named *def_pp* and *def_tree*, respectively. These sections are used to specify the arithmetic function calculated by the reference model. They correspond to the basic blocks depicted in Figure 1. These blocks are:

- a partial product generator, that determines the chosen Booth-encoding and computes the partial product with respect to this encoding
- an adder-tree, that will sum all partial products.

In the *def_pp* section, the encoding for partial products can be chosen. Furthermore, there is an option to specify a segmentation of the inputs. For this purpose, the language provides functions with the following signature:

- $Booth(b, r, h)$
- $PG(a, B, j)$

Given an input vector b, the radix $r \in \mathbb{N}$ and the position $0 \leq h \leq m+n-1$ for the *hot-one* bit, function $Booth(b, r, h)$ will calculate a bit vector B called *Booth-digit* for the selected encoding. As parameter for function $Booth$, b can be replaced by an arbitrary number of consecutive bits of b concatenated with constant values. Based on the resulting Booth digits B the function PG will calculate the partial products pp_j that are inputs to the addition part of the multiplier. Just like b also a can be replaced by slices a' concatenated with constants in function calls to PG.

In order to model typical optimizations in adder trees for booth encoded partial products, a parameter h is used in function $Booth$ to indicate the position of the so called *hot-one* or *hot-two* bit.

When generating the gate level reference each call to the functions and causes an instantiation of a corresponding generic VHDL entity.The arithmetic bit level description of the reference is generated by expanding the booth encoded partial products into weighted sums of bit wise multiplications $pp_j = pp_{1,j} + pp_{2,j} + pp_{3,j}$ as described in Section III-C.

A special treatment for the negative addends has been implemented to handle the optimizations performed on the addition tree due to sign extensions. To this end we now describe how to specify the addition tree of a multiplier-like function implementation in ARDL. Given timing and area constrains a designer will choose Addstep-, CSA-, Wallace trees, etc. to obtain an optimal implementation. For specifying such addition trees ARDL provides functions with the following signatures:

- $SUM(A)$ and
- $CARRY(A)$.

These functions map a set of vectors A to a single vector a. Since they are defined on sets of vectors, they can realize different tree structures with an arbitrary number of inputs to the nodes. The functions SUM and $CARRY$ are translated into their gate level representations in a straightforward way by instantiation of a CSA-Adder with corresponding outputs for sum and carry.

In order to obtain the arithmetic bit level representation for the adder-tree the intermediate results are recursively substituted by the arithmetic bit level description of their *fanin*. This process finally leads to a set of addition networks corresponding to the adders in the *def_tree* section. Note that the sets of addends for these addition networks will only contain partial products or primary inputs.

For the reference model, the equivalence of its gate level netlist and its arithmetic bit level description follows from the equivalence of the local function implementations.

Example 1 simple 4bit multiplier

```
variables {
   a: in (0 to 3);
   b: in (0 to 3);
   sum: out (0 to 7);
   carry: out (0 to 7);
   prod: out (0 to 7);
   ppb: pp(a,b) (0 to 2) }
pp_def{
   ppb(0) <= PG(a(0 to 3), Booth(0 & 0 & b(0),2,2),0);
   ppb(1) <= PG(a(0 to 3), Booth(b( 0 to 2),2,2),1);
   ppb(2) <= PG(a(0 to 3), Booth(b( 2 to 3) & 0,2,2),2) }
tree_def{
   s1a <= SUM(ppb(0), ppb(1), ppb(2));
   c1a <= CARRY(ppb(0), ppb(1), ppb(2));
   sum <= s1a;
   carry <= c1a;
   int_prod <= SUM(s1a, c1a);
   prod <= int_prod }
```

Algorithm 1 gives an example how a simple 4bit multiplier with radix 2 booth encoding is specified in the proposed language ARDL. This example defines an output for the final product and two additional outputs for the carry-save representation of the product.

E. Equivalence check

When checking the equivalence of the ARDL reference and the design under verification the reference model is compiled into VHDL and standard equivalence checking is employed. To keep the equivalence check tractable the following issues need to be given special consideration.

Equivalence checking is simpler if a large number of internal equivalences between the designs under comparison exist. To benefit from this, our description language is designed to achieve a high amount of internal equivalences. The circuit designer is requested to document the arithmetic algorithm implemented in the design using ARDL. This will lead to sufficient similarity between the reference multiplier and the design. In particular, the designer is requested to use the same topology of the adder tree for both, reference and implementation. This includes the order of inputs to the individual adders. This is essential as it is well known that swapping operands in addition trees is sufficient to eliminate most of the internal equivalences, thus increasing the proof complexity beyond the capacity of modern equivalence checkers.

Furthermore, we try to create as many equivalence as possible related to inputs of the adder tree. This is achieved by creating the same partial products which requires that the same kind of Booth encoding is used for reference model and design. Even more equivalences on the inputs of the adder trees are achieved if the correction signals *hot-one* or *hot-two* for sign conversion of negative values are added into the reference scheme in the same way as in the implementation. ARDL provides a mechanism to facilitate this.

Following the above guidelines, we obtain a high degree of structural similarity between reference and implementation. In our experiments nearly all intermediate results are equivalent between reference and design. Therefore, a structural analysis of both designs is usually sufficient to reduce the SAT instance by constant propagation and, even more importantly, by identical subexpression elimination [6].

The proof will either show that design and reference are equivalent or a counterexample will be calculated.

IV. Extension and Integration

In this section we will briefly discuss how the proposed approach can be integrated into existing verification methodologies and how the approach can be extended to solve a wider range of verification problems.

Arithmetic units of modern processor designs often support instructions for more complex tasks than multiplication of two single operands. For example, some designs contain hardware to accelerate matrix operations in order to enhance image processing. Some of these operations consist of several multiplications and additions but typically only one multiplier is realized in hardware. Nonetheless, if the multiplier is of sufficient width it can be reused to calculate several smaller products in parallel. To calculate the sum of those products additional shifters and adders are required in the design.

Our approach can also handle these complex operations. We use a distinct reference model for every embedded operation of this kind. For the reference, we reuse the addition network of the multiplier reference model created when verifying the multiplier in the design. We augment this addition network with additional adders in the same way as it has been done in the design. By slicing portions of the operands we obtain the appropriate encoding for the partial products.

A further extension of our approach can be obtained by integrating the reference multiplier in more complex reference

models for designs that use integer multipliers. For example, floating-point multiplication is performed on integer multipliers with additional logic for exponent handling and rounding. Approaches like [2] that black-box the multiplier could benefit from the structurally similar multiplier reference that allows us to do equivalence checking of a complete FPU reference model against the design. Thereby, as a side effect, the verification of the integer multiplier completes the FPU verification efforts in a full custom design flow.

V. Results

For evaluation, the proposed approach has been applied to verify several multiplier designs from IBM, created in a full custom design flow. In this section, we will present some details about these experiments. In order to show the flexibility of our approach we verified an integer multiplier with Booth-2 encoding for 24bit operands. The design also supports addition of two 32-bit products or four 16-bit products. For each of the operations a separate reference model has been created. The design implemented a mixture of different *hot-one* and *hot-two*-encodings and required different tree structures for each operation. We also tested integer multipliers for 53bit and 64bit operands. All designs implement a Wallace tree and are highly optimized on bit level, e.g. constant bits of partial products might be omitted. Furthermore, we present results of two experimental designs for 4bit and 8bit operands. All results where computed on a 64bit 2GHz Power5 machine.

operation (operand's width in bit)	cpu time AP	EC
4x4	0.6s	2s
8x8	1s	2s
8x8+8x8+8x8+8x8	9s	2s
16x16+16x16	9s	10s
24x24	7s	10s
53x53	8min	15s
64x64	14min	21s

TABLE I
Experiments runtime

The experimental results are reported in Table I where columns two and three show CPU-Times for the arithmetic proof (AP) and the equivalence check (EC). The results reflect the quadratic growth of the model's addition network. Note that our prototype implementation for the arithmetic check is based on a simple scripting language. Therefore, we believe that further optimizations of the implementation can reduce the CPU-time significantly.

The time necessary to prove equivalence between reference and design is also related to the width of the multiplier. In these experiments subexpression elimination was sufficient to solve the equivalence checking problems. Therefore, runtime increases with the number of expressions to be compared.

We believe that multiplier designs that can be described in the proposed language are contained in nearly every design of arithmetic circuits for IEEE754 floating-point and fixed-point operations. The manual effort to provide an ARDL reference is almost negligible compared the the overall design effort in a full-custom flow. While the datapath for multiplication has to be tested by simulation in other approaches our solution provides a formal proof of correctness. Hence, we can ensure high quality of the data path already in early design phases. This allows for a faster stabilization of the design throughout the design process, and possible changes like timing corrections can be verified immediately. Our methodology complements tests by simulation to focus on control structures of arithmetic circuits.

VI. Conclusions

In this paper we have proposed a new approach for practical multiplier verification based on a simple arithmetic proof in conjunction with equivalence checking. The key concept of our method is the construction of a reference model for the arithmetic circuit based on ARDL. The reference description serves two purposes. We can very easily extract the underlying arithmetic at the bit level and prove it by transforming it to a radix-2 representation. Furthermore, we can construct a structurally similar VHDL representation of the reference that implements the arithmetic described by the reference. By checking equivalence between reference and design we can decide whether the design is correct. The approach has proven to be easily extendable to meet different multiplier designs.

References

[1] R.E. Bryant: "Graph based algorithm for Boolean function manipulation"; IEEE Transactions on Computers, 1986, Page(s) 677-691, Vol C-35 No. 8

[2] C. Jacobi, K. Weber, V. Paruthi, J. Baumgartner; "Automatic Verification of Fused-Multiply-Add FPUs"; Design, Automation and Test in Europe 2005, Proceedings; Page(s) 1298-1303 Vol 2.

[3] Y.-T. Chang, K.-T. Cheng; "Induction-based gate-level verification of multipliers"; ICCAD 2001, Page(s) 190-193

[4] R. Bryant, Y. A. Cheng; "Verification of Arithmetic Functions by Binary Moment Diagrams"; Design Automation Conference 1995, Proceedings Page(s) 535-541

[5] D. Stoffel, W. Kunz; "Verification of integer multipliers on the arithmetic bit level"; ICCAD 2001; Page(s) 183-189

[6] A. Kuehlmann, V. Paruthi, F. Krom, M. K. Ganai; "Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification"; IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Dez. 2005, Page(s) 1377-1394 Vol 21

[7] Lamb, K.D.; "Use of Binary Decision Diagrams in the Modelling and Synthesis of Binary Multipliers"; ASIC Conference and Exhibit, 1996 Proceedings., 9th Annual IEEE International, 23-27 Sept. 1996, Page(s) 159 - 162

[8] T. Coe.; "Inside the Pentium Fdiv bug", Dr. Dobbs Journal, Apr. 1996, Page(s) 129-135

[9] S. Wefel, P. Molitor; "Prove that a faulty multiplier is faulty", 10th Great Lake Symposium on VLSI, 2000, Page(s) 43-46

[10] T. Stanion; "Implicit verification of structurally dissimilar arithmetic circuits", International Conference on Computer Design, 1999, Page(s) 46-50

[11] K. Hamaguchi, A. Morita, S. Yajima; "Efficient construction of binary moment diagrams for verifying arithmetic circuits", Proceedings of the International Conference on Computer Aided Design, Nov. 1995, Page(s) 78-82

[12] J.-C. Chen, Y.-A. Chen; "Equivalence checking of integer multipliers" Proceedings of the 2001 Conference on Asia South Pacific Design Automation, 2001, Page(s) 169-174

[13] M. Fujita; "Verification of arithmetic circuits by comparing two similar circuits", Proceedings of the 8th International Conference on Computer Aided Verification, 1996, Page(s) 159-168

[14] Y.-T. Chang, K. Ting; "Self-referential verification of gate-level implementations of arithmetic circuits", Proceedings of the 39th conference on Design automation, 2002, Page(s) 311-316

[15] M. Wedler, D. Stoffel, W. Kunz; "Normalization at the arithmetic bit level", Proceedings of the 42nd Design Automation Conference, Jun 2005, Page(s) 457-462

[16] M. D. Aagaard, C.-J. H. Seger; "The Formal Verification of a Pipelined Double-Precision IEEE Floating-Point Multiplier", Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design, Sept. 1995, Page(s) 7-10

[17] R. Kaivola, N. Narasimhan; "Formal verification of the Pentium 4 multiplier", Proceedings of the 6th IEEE International High-Level Design Validation and Test Workshop, 2001, Page(s) 115-120

[18] Jin Yang, Carl-Johan H. Seger, "Introduction to Generalized Symbolic Trajectory Evaluation," IEEE Transactions on Very Large Scale Integration Systems, June 2003, Page(s) 345-353