Self-referential verification of gate-level implementations of arithmetic circuits

Ying-Tsai Chang[*] and Kwang-Ting (Tim) Cheng
Department of Electrical and Computer Engineering
University of California
Santa Barbara, CA 93106
ytchang,timcheng@windcave.ece.ucsb.edu

ABSTRACT

Verification of gate-level implementations of arithmetic circuits is challenging due to a number of reasons: the existence of some hard-to-verify arithmetic operators (e.g. multiplication), the use of different operand ordering, the incorporation of merged arithmetic with cross-operator implementations, and the employment of circuit transformations based on arithmetic relations. It is hence a peculiar problem that does not fit quite well into the existing RTL-to-gate equivalence checking methodology. In this paper, we propose a self-referential functional verification approach which uses the gate-level implementation of the arithmetic circuit under verification to verify itself. Specifically, the verification task is decomposed into a sequence of equivalence checking subproblems, each of which compare circuit pairs derived from the implementation under verification based on the proposed self-referential functional equations. A decomposition-based heuristic using structural information is employed to guide the verification process for better efficiency. Experimental results on a number of implementations of the multiply-add units and the inner product units with different architectures demonstrate the versatility of this approach.

Categories and Subject Descriptors

B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*verification*

General Terms

Algorithm, Verification

Keywords

Arithmetic circuit verification

1. INTRODUCTION

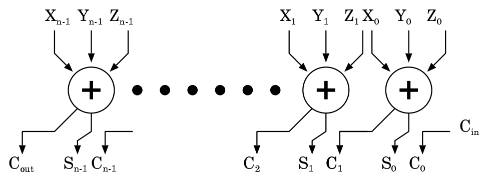
Combinational equivalence checking is a relatively mature field after over a decade's intensive research. Commercial combinational equivalence checking tools have advanced greatly in both speed and capacity, and have been successfully employed in the full-chip context for RTL functional validation sign-off. The key idea behind the solution to this intractable problem for complex industrial circuits is the effective use of structural similarities between circuits under comparison. Structurally similar circuits have an appreciable portion of functionally equivalent internal signal pairs between them, and these signal pairs could serve as cutpoints to facilitate the progression of equivalence checking in the miter approach [2]. With the aid of a robust RTL-to-gate equivalence checking tool, design functional validation and verification could be conducted more efficiently at the RTL level. A successful abstraction at a higher level of description could further facilitate design architecture exploration.

One class of circuits that proved to be difficult for RTL-to-gate equivalence checking is arithmetic circuits. Among them, the most familiar one is the multiplication operator. Arithmetic circuits play an important role in modern high-performance VLSI datapath design. The quest for performance in arithmetic circuits has stimulated the development of various architectures. However, these proliferated architectures have also posed new challenges to their implementation verification. Existing equivalence checking techniques are often incapable of handling some of these circuits due to either the lack of equivalent internal signal pairs when compared to a reference implementation with a different architecture, or the incapability of direct verification, e.g. BDD memory explosion or exponential number of backtracks for ATPG or SAT approaches. Since arithmetic operations are also the natural word level primitives in HDL, this issue is even more prevalent in modern RTL design methodology. Hence, the verification of gate-level implementations of arithmetic circuits is of practical importance. In this paper, we analyze this problem and propose a new and efficient solution. We believe that part of this problem has not been properly addressed before, and the proposed solution is the first to be able to efficiently perform gate-level verification for a large class of arithmetic circuits.

2. GATE-LEVEL ARITHMETIC CIRCUIT VERIFICATION - THE ISSUES

The difficulty of verifying gate-level implementations of arithmetic circuits is mainly due to the existence of some hard-to-verify constructs. These constructs include individual arithmetic operators (e.g. multiplication), the use of different operand ordering, merged arithmetic with cross-operator implementations, and circuit transformations based on arithmetic relations. Optimizations of arithmetic circuits for performance, which are often based on arithmetic relations, e.g. Booth's recoding or carry-save adder (CSA)

[*]Current affiliation: Novas Software, San Jose, CA 95110. `E-mail: ytchang@novas.com`



X_{n-1} Y_{n-1} Z_{n-1}
X_1 Y_1 Z_1 X_0 Y_0 Z_0
C_{in}
C_{out} S_{n-1} C_{n-1}
C_2 S_1 C_1 S_0 C_0

Figure 1: An n-bit carry-save adder.

making use of redundant number systems, inadvertently introduced some of these hard-to-verify constructs. In the modern HDL-based RTL design methodology, the capability of HDL design synthesis tools to perform some of these optimizations automatically and transparently further complicated their implementation verification problem.

The verification of arithmetic circuits has been addressed in a number of former researches, including backward construction of BMD [9], modular verification using BMD [6], implicit verification [16], residue BDD [11], the functional equation approach [8, 7], and the theorem-proving approach [1]. However, a more thorough analysis that across the entire spectrum of issues is still lacking. In the following, we list the key issues that are the major obstacles to an effective verification of arithmetic circuits. The listing is ordered from the more local and gate-level issues to the ones with an increasingly global and behavioral nature.

- *The difficulty of verifying individual arithmetic operators*

  It is well-known that the verification of multiplication operator is a hard one in the BDD-based verification paradigm due to its exponential spatial complexity in BDD [4, 5]. The difficulty of multiplication operator verification is closely related to the integer factorization problem [7]. Therefore it seems that this difficulty is quite independent of the verification paradigm. This difficulty is further complicated by the existence of multiple globally different architectures and, therefore, could not be easily verified at an equivalence checking setting. In the architectural space, the implementation of a multiplication operator could employ different addition reduction trees or different recoding schemes. Commonly encountered architectures of addition reduction trees include addition arrays (AA), carry-save-adder arrays (CS), Wallace tree (WA), and with Booth recoding (BR) [13, 6]. The division operator is another example that is known difficult to verify.

- *Operand ordering*

  The problem of operand ordering originates from the fact that implementations of arithmetic operators in general do not reflect the *commutative* and *associative* laws structurally. For example, the multiplication operator is in general implemented asymmetrically, i.e. in the addition reduction tree implementation, one operand is decomposed and the other one is kept intact in forming the partial products. Based on the commutative law $x \times y = y \times x$, if we use the second operand for decomposition, the implementation based on the left hand side (decomposing y) and the one based on the right hand side (decomposing x) will be significantly different in terms of circuit structure. Further considering the associative law $(x \times y) \times z = x \times (y \times z)$, the space for equivalent implementations of the same function is even larger. For example, $x \times y \times z$ could have 12 different implementations based solely on exploring the different ordering/grouping of the 3 operands.

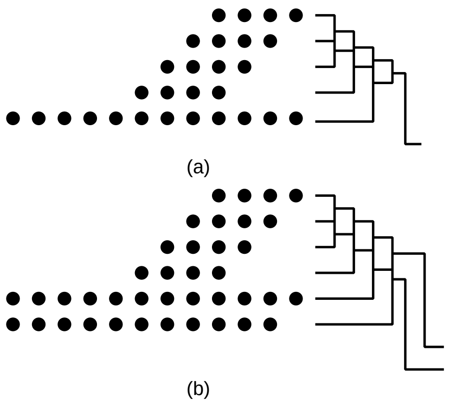


Figure 2: Two different multiply-add units implemented in carry-save-adder tree architecture.

- *Merged arithmetic with cross-operator implementation*

  Merged arithmetic refers to the direct implementation of multi-operator arithmetic expressions without breaking down into individual operators [17, 10, 18]. This approach often employs arithmetic relations to achieve a cross-operator implementation with a single addition reduction tree [14]. Merged arithmetic can achieve higher speed by reducing the number of carry propagation chain, and often has a smaller circuit size and lower power consumption. However, due to the lack of intermediate signals representing the individual operator decomposition, merged arithmetic circuits are an especially hard class of circuits for functional verification.

  One example is the multiply-add unit commonly encountered in the DSP design. To reduce the delay of the critical path, the multiply-add operation is implemented as one single addition reduction tree, i.e. the addition operator is merged into the addition reduction tree of the multiplication operator. This is often achieved by employing the CSA array (Figure 1). Implementations of two types of 4-bit multiply-add units based on the CSA array architecture are shown in Figure 2. In this figure, we use the dot notation and their skeleton addition reduction tree structure to represent the circuit implementation. The dots denote the bits of addenda in the addition reduction tree. The 3-to-2 and 2-to-1 reduction symbols represent the CSA and the binary full adder respectively. In multiply-add units, the addend typically has a larger size to prevent overflow or to reduce accumulation error. Figure 2 (a) implements the multiply-add function $S = a \times b + c$ directly, and Figure 2(b) implements another type of multiply-add unit with the addend and result in the CSA form for faster accumulation at successive operations, i,e, $S_0 + S_1 = a \times b + c_0 + c_1$. This figure shows two different multiply-add units implemented using the same CS architecture. In general, any addition reduction tree architecture utilized in the implementation of multiplier operators could also be used in merged arithmetic. Merged arithmetic hence introduced another rather large degree of freedom in the design implementation space.

  Another example, where this problem is more serious, is the inner product unit, e.g. $S = a_0 \times b_0 + a_1 \times b_1$ and higher di-

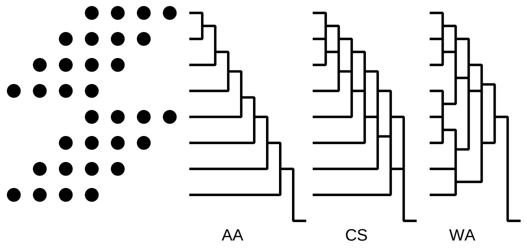


Figure 3: Designs of the inner product unit with different addition reduction tree structures

mensional generalizations. As shown in Figure 3, which follows the convention in Figure 2, the inner product unit could also have a number of different implementations in which the two multiplication operators are implemented in one single addition reduction tree. In this case, even the intermediate products are missing from the implementation, and its verification demands a new approach. The main issue for merge arithmetic is that operations of redundant number representation, such as CSA, do not have a suitable RTL description in modern HDL. Functional equivalence checking in this context is hence very difficult.

- *Circuit transformation by arithmetic relations*

  At a more global level, arithmetic circuits could be simplified by explicitly using more arithmetic transformations, such as the *distributive* law $x \times (y+z) = x \times y + x \times z$. Architectural changes by exploring common expression factoring and expansion often significantly change the datapath, e.g. $f = a-b,\ g = a+b,\ h = a^2 - b^2 \to h = fg$. Verification of circuits employing such transformations seems to be more tractable only at the RTL level [19]. Compared to the associative and commutative laws, the circuit transformations based on the distributive law is more difficult to prove, presumably because the distributive law bridges between two different arithmetic operations.

- *Floating point units.*

  The verification of floating point units as a class of arithmetic circuits is further complicated by the issue of rounding schemes and operand alignment. This class of circuits seems to be more suitably addressed with the assistance of the theorem-proving techniques [1], which requires more architectural information and human intervening. We do not consider this problem in this paper due to our main interest in the more common arithmetic operators encountered in modern HDL.

The above issues seriously impair our capability of gate-level arithmetic circuit verification at the common RTL-to-gate equivalence checking setting. In this paper, we present a self-referential functional verification approach which properly addresses the first three issues.

3. SELF-REFERENTIAL FUNCTIONAL VERIFICATION

The basic idea of self-referential functional verification is to use the gate-level implementation of the arithmetic circuit under verification to verify itself. More specifically, we decompose the arithmetic circuit verification problem into a sequence of mathematical equations, which entail the correctness of the arithmetic circuit when established. These mathematical equations are verified by checking the equivalence of similar circuit pairs derived from the circuit under verification based on these equations. Notice that the difficulty of gate-level arithmetic circuit verification is mainly due to the proliferation of globally different architectures of arithmetic circuits originated from the existence of arithmetic relations, while self-referential functional verification approach explicitly utilizes these arithmetic relations to facilitate their verification instead.

3.1 Verification by exploiting functional equations

Consider the verification of a circuit F implementing a binary arithmetic function f composed of $+,-$ and $\times$ operators in an expanded form, i.e. sum $(+,-)$ of product form as an arithmetic expression, with distinct l_i-bit binary operands x_i. The operands x_i of f are composed of Boolean bit variables x_{ij} such that the binary representation of x_i is $(x_{i(l_i-1)}x_{i(l_i-2)}..x_{i0})_2$. The total number of Boolean bit variables in f is L, i.e. $L = \sum_i l_i$. An ordering $(y_1,y_2...,y_L)$ of all bit variables $\{x_{ij}\}$ defines a sequence of sets $(A_0,A_1..,A_L)$ with A_k composed of the first k elements in this ordering, i.e. $A_0 = \phi$, and $A_k = A_{k-1} \cup \{y_k\}$. For $y_k = x_{ij}$, we use the notation $\tilde{y_k}$ to denote the set of bit variables composed of all x_{im} with $m \neq j$ and all bit variables of the operands that are not within the same product term with x_i in f, e.g. for $f = x_1 \times x_2 + x_3 \times x_4$, the set $\tilde{x_{12}}$ is composed of all $x_{1m} (m \neq 2)$, x_{3n}, and x_{4n}. We also use the notation $f|_{A=0}$, to denote the function f with the values of the bit variables in A restricted to zero. We have the following:

THEOREM 1. *(Soundness) For any ordering $(y_1,y_2..,y_L)$ of all bits $\{x_{ij}\}$, consider the finite sequence of sets $A_0...,A_L$ associated with this ordering as defined above. Then the set of $2L$ equations $(k = 1,2..,L)$:*

$$F|_{A_{k-1}=0} = F|_{A_k=0} \pm F|_{(\tilde{y_k} \cup A_{k-1})=0} \quad (1)$$
$$F|_{(\tilde{y_k} \cup A_k)=0} = f|_{(\tilde{y_k} \cup A_k)=0} \quad (2)$$

where $\pm$ depends on whether the term containing $\tilde{y_k}$ in f is $+$ or $-$, and $F|_{A_L=0} = f|_{A_L=0}$ implies $F = f$.

PROOF. It follows from the definition of binary number representation, the distributive law, and induction on the total number of nonzero bits in all operands. □

Our self-referential functional verification of arithmetic circuits is based on this theorem. Notice this theorem is on a specific form of f rather than the function itself. However, even with this restriction, as will be shown in examples below, Theorem 1 still includes a lot of interesting cases. In essence, this theorem uses the distributive law to decompose the binary digits inside the arithmetic product terms of the mathematical expression f, and hence also the circuit F. For a gate-level implementation F of an arithmetic circuit f, the equations in (1) are verified by equivalence checking the circuits implemented at both sides of the equations, and the equations in (2) are verified by invoking Theorem 1 again on the new function $f|_{(y_k \cup A_k)=0}$. Note that the implementations of both sides of the equation (1) are derived from the same original circuit under verification F.

Intuitively, this sequence of equations successively reduces the original verification problem into the ones with smaller nontrivial circuit sizes and fewer inputs by bit-decomposition. Notice that the specification function f only enters in the form of specifying $\tilde{x_{ij}}$, $\pm$, and terminal cases, hence it is a framework of verification without an implementation standard. Although seemingly two copies of different F's appear in the right hand side of equation (1), as will be discussed below, a judicious choice of the ordering $y_0...,y_l$ will result in significant isomorphic portion in these two circuits that could be shared between them. Thus, the equivalence checking problems represented by the above equations are in fact comparing two F's with small difference in peripheral circuits and constraints; hence the name *self-referential* verification.

For example, to verify whether $F(x_{n-1}..,x_0,y_{n-1}..,y_0)$ implements $f = x \times y = (x_{n-1}x_{n-2}...x_0)_2 \times (y_{n-1}y_{n-2}...y_0)_2$, it suffices to prove the following equations $(k = 0,1,..n-1)$:

$$\begin{aligned}
& F(0..0,x_{n-1-k},..,x_0,y_{n-1}..,y_0) \\
= \; & F(0..0,x_{n-2-k}..,x_0,y_{n-1}..,y_0) \\
& +F(0..0,x_{n-1-k},0..,0,y_{n-1}..,y_0) \\
& F(0..0,y_{n-1-k}..,y_0) \\
= \; & F(0..0,y_{n-2-k}..,y_0) \\
& +F(0..0,y_{n-1-k},0..0) \\
& F(0..0,x_{n-1-k},0..,0,y_{n-1}..,y_0) \\
= \; & f(0..0,x_{n-1-k},0..,0,y_{n-1}..,y_0) \\
& F(0..0,y_{n-1-k},0..0) \\
= \; & f(0..0,y_{n-1-k},0..0)
\end{aligned}$$

and $F(0..,0) = f(0..,0)$ on F. Notice that in this special case the ones with bit variables in one operand all equal to zero are often trivial to prove due to the fact that $0 \times y = 0$. To put it in a more familiar form, this sequence of equations essentially verifies the following equation for different k on F:

$$\begin{aligned}
& (0..0x_{n-1-k}x_{n-2-k}..x_0)_2 \times (y_{n-1}..y_0)_2 \\
= \; & (0..0x_{n-2-k}x_{n-3-k}..x_0)_2 \times (y_{n-1}..y_0)_2 \\
& +(0..0x_{n-1-k}0..0)_2 \times (y_{n-1}..y_0)_2
\end{aligned}$$

The verification of arithmetic circuits posed as equivalence checking problems also has the benefit of gracefully handling design error. A number of previous approaches were incapable of handling design error. In particular, word-level decision diagram based techniques [6, 9] share this problem due to the explicit word-level feature, which makes them incapable of representing bit-level errors appearing in a gate-level netlist. However, when posed in the equivalence checking setting, modern equivalence checking methodology is often capable of finding a counterexample for inequivalence quickly, and thus providing a convenient solution to this design error problem.

This proposed approach could not be easily applied to the division operator due to the fact that the decomposition functional equation for the division operator requires the information of the remainders, i.e. $(x+y)/z = x/z + y/z + (x\%z + y\%z)/z$, which might not be explicitly available in the circuit implementation under verification. Although it would still be desirable to be able to handle these cases, the division and remainder operations are not as often used in modern HDL design as the other operations handled by our approach.

3.2 Expediting equivalence checking problems

Efficient execution of these equivalence checking problems could in general be achieved by judiciously choosing the ordering $(y_1...,y_L)$ for bit-decomposition using structural information:

OBSERVATION 1. *(Effectiveness I) In verifying the sequence of equations in Theorem 1, choosing the bit with the smallest fanout cone size for decomposition first will result in easier equivalence checking problems.*

The fanout cone size of a signal denotes the number of signals within the fanout cone of that signal in the circuit. With this choice of bit-decomposition, the resulting equivalence checking problem will have a lot more equivalent internal signal pairs between the two circuits under equivalence checking. In fact, the resulting equivalence checking problem could be performed by first sharing common circuit portions between $F|_{A_{i+1}}$ and $F|_{\tilde{y}_i \cup A_i}$, and then merging the isomorphic portions between both sides of the equation in the miter approach. The equivalence checking problem is then effectively reduced to verifying the equivalence of small portions of the two circuits under the constraint that these portions are driven by the merged isomorphic portion.

Choosing the bit for decomposition from the ones with small fanout cone sizes could also help the design error diagnosis. For example, if the largest i for which the equation in Theorem 1 is not satisfied is α, then there is an error in the fanout cone of y_α within circuit $F|_{A_{\alpha-1}=0}$. Notice that the operand ordering problem and the issue of merged arithmetic are automatically addressed by this decomposition strategy.

Although the application of theorem 1 could involve recursive invocation due to the requirement of verifying equation (2), it is often possible to reuse former equivalence checking results. For example, to verify whether $F(x_{n-1}..,x_0,y_{n-1}..,y_0,z_{n-1}..,z_0)$ implements $f = x \times y \times z = (x_{n-1}x_{n-2}...x_0)_2 \times (y_{n-1}y_{n-2}...y_0)_2 \times (z_{n-1}z_{n-2}...z_0)_2$ with ordering $(x_{n-1}..,x_0,y_{n-1}..,y_0,z_{n-1}..,z_0)$, it requires to prove the following equations $(k = 0,1,..n-1)$:

$$\begin{aligned}
& F(0..0,x_{n-1-k},..,x_0,y_{n-1}..,y_0,z_{n-1}..,z_0) \\
= \; & F(0..0,x_{n-2-k}..,x_0,y_{n-1}..,y_0,z_{n-1}..,z_0) \\
& +F(0..0,x_{n-1-k},0..,0,y_{n-1}..,y_0,z_{n-1}..,z_0) \\
& F(0..0,y_{n-1-k}..,y_0,z_{n-1}..,z_0) \\
= \; & F(0..0,y_{n-2-k}..,y_0,z_{n-1}..,z_0) \\
& +F(0..0,y_{n-1-k},0..0,z_{n-1}..,z_0) \\
& F(0..0,z_{n-1-k}..,z_0) \\
= \; & F(0..0,z_{n-2-k}..,z_0) \\
& +F(0..0,z_{n-1-k},0..0)
\end{aligned}$$

and the following recursive invocation of Theorem 1:

$$\begin{aligned}
& F(0..0,x_{n-1-k},0..,0,y_{n-1}..,y_0,z_{n-1}..,z_0) \\
= \; & f(0..0,x_{n-1-k},0..,0,y_{n-1}..,y_0,z_{n-1}..,z_0) \\
& F(0..0,y_{n-1-k}..,y_0,z_{n-1}..,z_0) \\
= \; & f(0..0,y_{n-1-k}..,y_0,z_{n-1}..,z_0) \\
& F(0..0,z_{n-1-k},0..0) \\
= \; & f(0..0,z_{n-1-k},0..0)
\end{aligned}$$

These equations could be simplified by the equations:

$$\begin{aligned}
& F(0..0,x_{n-1-k},0..,0,y_{n-1}..,y_0,z_{n-1}..,z_0)|_{x_{n-1-k}=x_{n-1-k-l}} \\
= \; & F(0..0,x_{n-1-k-l},0..,0,y_{n-1}..,y_0,z_{n-1}..,z_0) \times 2^l
\end{aligned}$$

which will often result in more efficient equivalence checking in most implementations, and reduce to only one recursive invocation of theorem 1 in this case.

The appearance of possible false negatives in the direct application of the previous prescription should be properly addressed. It was pointed out in [7] that cutpoints inside a full adder is an intrinsic source of false negative. It could be avoided using the fanout cone of the adjacent bit in the ordering to move the support backward. Care should be taken to handle these false negative systematically to avoid excessive backward substitution in BDD-based equivalence checking.

3.3 Implicit verification of arithmetic circuit

One particularly serious problem for Wallace tree architecture using BDD-based equivalence checking in this approach is the large support variable size, which will result in excessive variable reordering. We found that the implicit verification approach [16] is capable of reducing the support variable size.

Implicit verification of arithmetic circuits is based on the fanin-based partitioning over all primary outputs of the arithmetic circuit [16]. It was observed that, under the setting of the equivalence checking of arithmetic circuits, the equivalence of lower significant bits could be used as a precondition to facilitate the equivalence checking of the next higher bit. More specifically, given two arithmetic circuits with binary value outputs $c = (c_{l-1}c_{l-2}..c_0)_2$ and $c' = (c'_{l-1}c'_{l-2}..c'_0)_2$, the equivalence checking of these two circuits starts from the least significant bits (LSB) c_0 and c'_0. The equivalence checking of the more significant bits c_k and c'_k are conducted by checking whether the expression

$$(c_k \oplus c'_k)\overline{(c_{k-1} \oplus c'_{k-1})}...\overline{(c_{k-b_k} \oplus c'_{k-b_k})}$$

equals to zero for each k, where the parameter b_h increases successively for an individual k if the equivalence is not proved until the LSB is reached. In proving this proposition using local BDD-based approach, c_k and c'_k are built using signals at the fanin cone boundary of c_{k-1} and c'_{k-1} as support variables, and the expressions above involving different contiguous bits are evaluated using composition. This prescription of local BDD construction could potentially avoid the intermediate explosion of BDDs corresponding to the above expression, and it turns out to be crucial in handling Wallace tree architecture.

4. EXPERIMENTAL RESULTS

Experimental results for different implementations of the following three arithmetic functions are shown in Table 1:

$$\begin{aligned} I: &\quad S = a \times b + c \\ II: &\quad S = a_0 \times b_0 + a_1 \times b_1 \\ III: &\quad S = a \times b \times c \end{aligned}$$

where a's and b's are of size n bits, and c's are of size $3n$ bits in I and n bits in III. Some implementations of these arithmetic functions with different architectures are verified using the proposed approach with no *a priori* architectural information given to the verification program. The architectures are denoted in the subscripts using the abbreviations introduced in section 2. Function III essentially has two addition reduction tree and we denote their architectures in the subscript sequentially. The experiments are performed on a Pentium III 733MHz machine with 256MB memory running Linux OS. The results are shown with the upper field representing the verification time in seconds and the lower field representing memory usage in MB. The number appearing in the first row denotes the size parameter n within the corresponding column. The results for circuit size up to $n = 80$ is shown. A BDD-based equivalence checking procedure using CUDD package [15] with sifting reordering is used. We used implicit verification [16] to effectively reduce the support variable size. The results show that our approach is capable of verifying these examples with different architectures. The only one that we could not finish is I_{WA} for $n = 80$, which still suffers from excessive variable reordering even with the implicit verification technique. Notice that no prior work could be compared to our result due to the minimal assumptions made on the circuits in our approach. However, our results do show better or comparable performance to other former approaches even with weaker assumptions on the circuits. The low memory usage also suggests that structural similarities in each equivalence checking problem are effectively utilized. Our results support the following observation:

n	16	32	48	64	80
I_{AA}	0.96	16.7	86	258	617
	0.61	0.92	1.54	1.60	2.21
I_{CS}	1.59	17.7	87	259	619
	0.65	0.95	1.60	1.65	2.01
I_{WA}	1.72	42	251	1720	
	0.61	0.65	0.87	0.91	
I_{BR}	3.61	47	261	780	2476
	0.58	0.75	3.98	4.06	4.58
II_{AA}	1.87	33	161	503	1192
	0.41	0.71	1.02	1.23	1.61
II_{CS}	2.21	35	171	502	1196
	0.43	0.70	1.05	1.40	1.68
$III_{AA,AA}$	1.89	33	183	502	1215
	0.42	0.72	0.97	1.34	1.63
$III_{CS,CS}$	2.30	35	174	513	1237
	0.45	0.75	1.12	1.42	1.69

Table 1: Experimental results. Within each entry, the upper number is the verification time in seconds and the lower one is the memory usage in MB. (abbreviations: Addition array(AA), Wallace tree (WA), Carry-save-adder array (CS), Booth's recoded (BR))

OBSERVATION 2. *(Effectiveness II) Common architectures of arithmetic circuits have a bit-decomposition ordering that could result in effective execution of the equivalence checking problems in Theorem 1.*

This observation is a reflection of the underlying addition reduction tree structure. Theorem 1 by itself is merely a mathematical statement. Only the evidence from Observations 1 and 2 could justify the effectiveness of the proposed approach.

5. RELATED WORK

Verification of arithmetic circuits using functional equations was first introduced in [8], where the equation $x \times (y+1) = xy + x$ is used for the verification of multipliers. The verification problem of one multiplier is then posed as one single functional equation, but it is executed as a sequence of equivalence checking problems to break the carry-propagation chain in the $(y+1)$ term by cases. However, the circuits under equivalence checking could have quite few equivalent internal signal pairs due to the large fanout cone of the difference circuit, and the resulting equivalence checking problems could be quite hard.

Functional equation has also been used to verify multipliers in [7] with a similar procedure described in this paper. The solution proposed in the current paper offers a more general framework that could handle a larger class of arithmetic circuits through a structural-based bit-decomposition strategy. The inclusion of the implicit verification technique further extends the capability of this framework to handle Wallace tree architecture, which could not be handled effectively in [7].

Implicit verification [16] is the first technique that could handle equivalence checking over structurally dissimilar arithmetic circuits. In our context, the circuits under comparison are structurally similar, but implicit verification technique still introduced the benefit of reducing the support variable size, which is essential in some cases, e.g. Wallace tree architecture.

Self-referential verification is a novel idea that utilizes functional equations to use a circuit implementation to verify itself without referencing to a standard. However, similar idea has been proposed in quite different contexts that represent some even more surprising and elegant solution [12].

6. CONCLUSIONS AND FUTURE DIRECTIONS

We present a systematic analysis of the gate-level arithmetic circuit verification problem and propose a self-referential approach to solve this problem. Experimental results show that this approach is capable of handling a number of arithmetic circuits with different architectures that cannot be verified effectively by existing approaches, including the multiply-add units and the inner product units. This framework is capable of handling most arithmetic circuits composed of operators $+, -$, and $\times$ with different operand ordering and even with merged arithmetic implementations. It would be useful to integrate this approach into a generic equivalence checking engine.

This framework seems to be incapable of handling circuits derived by employing transformations using arithmetic relations other than the commutative and associative laws, especially factorization and expansion associated with the distributive law. We believe this problem should be best solved at the RTL level with the integration of a computer algebraic system [3]. Only a tight integration of an equivalence checker, a gate-level arithmetic circuit verifier and a computer algebra system could completely solve all the challenging issues of arithmetic circuit verification problem in modern HDL-based VLSI design.

7. ACKNOWLEDGEMENTS

This research was supported in part by SRC Task 835.001 and NSF International Research Center for SoC. The final version of this paper was prepared when the first author is at Novas Software.

8. REFERENCES

[1] Special issue on verification of arithmetic hardware. *Formal Methods in System Design*, 14(1), 1999.

[2] D. Brand. Verification of large synthesized designs. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 534–537, November 1993.

[3] R. K. Brayton. The future of logic synthesis and verification. In *Logic Synthesis and Verification*, pages 403–434. Kluwer Academic Publishers, 2002.

[4] R. E. Bryant. Graph-based algorithm for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, 1986.

[5] R. E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Trans. on Computers*, 40(2):205–213, 1991.

[6] R. E. Bryant and Y. A. Chen. Verification of arithmetic circuits with binary moment diagrams. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference (DAC)*, pages 535–541, 1995.

[7] Y.-T. Chang and K.-T. Cheng. Induction-based gate-level verification of multipliers. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, pages 190–193, November 2001.

[8] M. Fujita. Verification of arithmetic circuits by comparing two similar circuits. In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV)*, pages 159–168, 1996.

[9] K. Hamaguchi, A. Morita, and S. Yajma. Efficient construction of binary moment diagrams for verifying arithmetic circuits. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, pages 78–82, November 1995.

[10] T. Kim, W. Jao, and S. Tjiang. Arithmetic optimization using carry-save-adders. In *Proceedings of the 35th ACM/IEEE Design Automation Conference (DAC)*, pages 433–438, 1998.

[11] S. Kimura. Residue BDD and its application to the verification of arithmetic circuits. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference (DAC)*, pages 542–545, 1995.

[12] A. Migdall. Correlated-photon metrology without absolute standards. *Physics Today*, 52(1):41–46, 1999.

[13] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 1999.

[14] M. Potkonjak and J. Rabaey. Optimizing resource utilization using transformations. *IEEE Trans. on CAD*, 13(3):277–292, 1994.

[15] F. Somenzi. *CUDD: CU decision diagram package Release 2.3.1*. University of Colorado at Boulder, 2001.

[16] T. Stanion. Implicit verification of structurally dissimilar arithmetic circuits. In *Proceedings of IEEE International Conference on Computer Design (ICCD)*, pages 46–50, 1999.

[17] E. E. J. Swartzlander. Merged arithmetic. *IEEE Transactions on Computers*, 29(10):946–950, 1980.

[18] J. Um, T. Kim, and C.-L. Liu. Optimal allocation of carry-save-adders in arithmetic optimization. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, pages 410–413, November 1999.

[19] Z. Zhou and W. Burleson. Equivalence checking of datapaths based on canonical arithmetic expressions. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference (DAC)*, pages 546–551, 1998.