Automatic Verification of Arithmetic Circuits in RTL Using Stepwise Refinement of Term Rewriting Systems

Shobha Vasudevan, Vinod Viswanath, Robert W. Sumners, and Jacob A. Abraham, *Fellow*, *IEEE*

Abstract—This paper presents a novel technique for proving the correctness of arithmetic circuit designs described at the Register Transfer Level (RTL). The technique begins with the automatic translation of circuits from a Verilog RTL description into a Term Rewriting System (TRS). We prove the correctness of the designs via an equivalence proof between TRSs for the implementation circuit design and a much simpler specification circuit design. We present this notion of equivalence between the TRSs and a stepwise refinement method for its decomposition, which we leverage in our tool Verifire. We demonstrate the effectiveness of our technique by using the tool for the verification of several multiplier designs that have hitherto been impossible to verify with existing approaches and tools.

Index Terms—Verification, Hardware Description Languages, Register Transfer Level implementation, arithmetic logic unit.

1 Introduction

As designs increase in complexity, verifying their correctness is becoming an increasing portion of the design effort; in some cases, verification involves more than half of the design effort. Among verification technologies, simulation is used to check the entire design, whereas formal approaches are being used to guarantee the correctness of some of the modules in a large design.

Formal verification of digital hardware can be classified broadly into two categories—state space-based techniques and deductive techniques. State space-based techniques like model checking [22], BDD-based verification [6], [9], and so forth, reason with the state space of the entire system at the Boolean level. In contrast, deductive verification techniques like theorem proving [25], [32], rewriting [24], and so forth, try to solve the verification problem by equational reasoning. Due to the high computational complexity of the verification problem, computer-aided verification methods are all partial or incomplete. In the case of automatic state space-based methods, this incompleteness manifests as a *state space explosion*, leading to practical time and space limitations. In the case of deductive methods that circumvent the state space explosion and are size independent, the incompleteness manifests itself as a lower degree of automation, requiring manual intervention during the verification process. Therefore, whereas state space-based approaches cannot handle circuits of even reasonable sizes, deductive verification approaches involve a significant manual component. Despite their incompleteness, theorem provers are widely used for the verification of complex hardware systems due to their efficiency in handling real designs and the high degree of confidence they provide.

Verification of integer arithmetic circuits has been a widely studied problem. Verification of relatively less complex designs like adders and shifters is an achievable target by state-of-the-art tools and techniques. However, in the case of complex arithmetic circuits, current verification methods do not scale. In particular, integer multipliers present a major challenge to verification tools and techniques. These circuits can be prohibitively expensive to analyze with Boolean-level algorithms that undertake explicit state space traversal [7], [8], [9], [13], [19], [39]. State-of-the-art equivalence checkers that use state space-based techniques and Boolean reasoning cannot verify multipliers larger than 16×16 due to their large state space. Specialized techniques like Binary Moment Diagrams (BMDs) [8] can verify big word-sized multipliers but are not generic in their scope. Although deductive verification techniques can handle the multiplier state space, a high level of user expertise is required to handle the engine and considerable effort is spent in proving many ancillary lemmas before proving the main theorem. A multiplier proof, for instance, can take many person-days to accomplish.

We propose a highly automated deductive verification technique for formally verifying fixed-point combinational arithmetic circuits, including multipliers at the Register Transfer Level (RTL). Our technique involves the stepwise refinement of Term Rewriting Systems (TRSs) [26]. We focus on complex circuits that are otherwise difficult to verify. Our technique retains the efficiency and the size independence of deductive verification techniques while sacrificing automation minimally. We present a dedicated arithmetic circuit checker as opposed to a generic rewriting engine that involves considerable user interaction. Arithmetic circuits have sufficient structural regularity to afford analysis by functional decomposition and are therefore ideal candidates for our verification by stepwise refinement.

• *The authors are with the Computer Engineering Research Center, University of Texas at Austin, 1 University Station C8800, Austin, TX 78712-0323. E-mail: {shobha, vvinod, sumners, jaa}@cerc.utexas.edu.*

Manuscript received 29 May 2004; revised 30 Mar. 2006; accepted 13 Apr. 2006; published online 7 May 2007.
Recommended for acceptance by S.J. Upadhyaya.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0184-0504.
Digital Object Identifier no. 10.1109/TC.2007.1073.

We prove the equivalence of an implementation (or revised) Verilog RTL design against a specification (or golden) Verilog RTL design. We translate the golden and revised combinational Verilog RTL designs into TRSs. We then prove input/output equivalence between the two TRSs. This notion of equivalence is compositional and thus also affords proof decomposition through stepwise refinement. In order to decompose this proof, we compute a set of *comparison points*.

Our tool, Verifire, is a dedicated arithmetic circuit checker that provides automatic support for the proposed technique. The tool translates the golden and revised designs from Verilog source into TRSs. It automatically generates the comparison points for the incremental equivalence proof of the two TRSs. It uses an incomplete but efficient method to generate equivalent proofs at the comparison points.

Arithmetic circuits are often designed and optimized incrementally from a base design. This can afford the development of generically applicable decomposition strategies for equivalence proofs of the optimized design against the base design. We have found this to be the case for several families of integer arithmetic circuit designs and have leveraged this intuition in our tool to verify the optimized design against the base design. We have successfully applied our technique to the verification of optimized adders, comparators, shifters, and multipliers.

Since the verification of multipliers provides an interesting nexus of challenge and opportunity, we make multiplier verification the main focus of this paper. We split the space of arithmetic designs into standard (base) designs and modified (optimized) designs. Standard designs are widely used designs. In multipliers, standard designs are Booth, Wallace Tree, and Array multipliers. We prove the equivalence of optimizations to these designs against the standard designs and prove the equivalence of the standard designs against a simple golden shift-and-add multiplier. To illustrate our technique, we present the proof of correctness of a nontrivial example, a 128×128 Booth multiplier [5], verified against a shift-and-add multiplier. We also outline the steps for proving the correctness of BISMUL [38], an optimized Booth multiplier against the normal Booth multiplier. A comparison of our approach with existing Boolean equivalence checkers shows that it has the ability to verify multipliers much larger than currently possible.

All of the analysis performed by the tool is done on *terms* composed of RTL operators (for example, bitwise-and, left shift, and so forth) as opposed to the Boolean netlist level —the level at which equivalence checkers operate. Terms are more concise and more efficient to manipulate than netlists. The potential downside is the incompleteness of the equivalence prover, but we have found that, in practice, sufficiently complete provers are reasonable to implement.

Our main contributions in this work are listed below:

- We present a dedicated arithmetic circuit checker, as opposed to a generic rewriting engine. Our tool operates on the actual RTL circuit implementation and generates proofs at that level of detail. There is minimal overhead of creating an environment, proving ancillary lemmas, and so forth, as opposed to a generic rewriting engine, thereby yielding savings of many person-months on complex proofs.
- Our experimental results are presented on large multipliers that are very complex circuits for verification. We demonstrate that our technique can perform equivalence checking between two diverse multiplier designs, irrespective of size, thereby showing performance benefits of many orders of magnitude than widely accepted industrial methods.
- We extend our technique to verify incremental modifications or optimizations to existing designs, thereby covering a large portion of the design space.
- We present a novel decomposition strategy for RTL equivalence checking. In our technique, comparison points are computed automatically by the equivalence checker.
- We define and use a novel notion of decomposed TRS equivalence.
- We present a methodology to automatically determine the decomposition in the TRS equivalence proof.

Section 3 explains our technique in detail. We illustrate our algorithm with a nontrivial example of adders in Section 3.4. We present our tool, Verifire, in Section 3.5. We provide a theoretical definition of TRS equivalence and our proof structure in Section 4. In Section 5, we provide a detailed correctness proof for the Booth multiplier and outline the proof for BISMUL. We also provide the results of comparing our tool against commercial equivalence checkers for large multiplier designs. We discuss other multiplier verification techniques and conclude in Section 6.

2 Background

We briefly review several definitions and concepts about term rewriting. A TRS is a set of rewrite rules where a rewrite rule is an ordered pair of terms denoted as $(t_1 \rightarrow t_2)$ with $\rightarrow$ pointing in the direction of the rewrite. A TRS is *terminating* if there are no infinite rewrite sequences $t_1 \rightarrow t_2 \rightarrow \ldots$. A TRS is *confluent* if any divergence in rewriting is eventually joined. A *normal form* is a term that cannot be rewritten any further. Termination ensures the existence of normal forms, whereas confluence ensures their uniqueness.

TRSs have been used in the past for program verification [2], [3], [18]. In the context of hardware, rewriting strategies have been used in the past to design correct circuits [4], [21], [34], [35]. TRSs were first proposed for hardware verification in [10]. Subsequently, they have been used for checking the functional correctness of hardware [30], [40].

3 The Algorithm

Our goal is to prove the equivalence of an implementation and a specification design. We assume that both of these designs are (or can be translated into) combinational Verilog RTL modules that define a function from their inputs to outputs. Therefore, the equivalence of these functions is the target of the analysis. Whereas traditional

```
main (vG, vR) {
    trsG := translate (vG)
    trsR := translate (vR)
    proof_outcome := prove (trsG, trsR)
}

prove (trsG, trsR) {
    CP := computeComparePoints (trsG, trsR)
    for (every comparison point (cG, cR) ∈ CP)
        if (reduce (cG) is not equal to reduce (cR))
            return failure
    return success
}

reduce (t) {
    while (some rule can be applied)
        rewrite (t)
}
```

Fig. 1. The algorithm.

combinational equivalence checking tools also analyze the same problem at the gate level, our analysis is at the RTL.

The monolithic verification problem is intractable in general and we use signal[1] names in the two modules as a guide in decomposing this equivalence proof.

The algorithm for our technique is presented in Fig. 1. The golden and revised Verilog designs are represented by *vG* and *vR*, respectively. A mapping of input and output signal names between the two designs is provided.

The *translate()* function translates the Verilog into a TRS —the resulting TRS will be termed a *structural* TRS. The details of the translation of Verilog designs to structural TRSs can be found in Section 3.1. *trsG* and *trsR* therefore represent the structural TRSs of the corresponding Verilog design. The *prove* function is invoked for checking the equivalence of the two TRSs. We decompose the equivalence proof by using *comparison points* for matching the TRSs. The function that computes these comparison points is *computeComparePoints()* (explained later, in Section 3.2). For each comparison point *cG* in the golden TRS, the corresponding point in the revised TRS is *cR*. The *reduce()* function simplifies the terms at the comparison point by applying a set of rewrite rules. The simplification is done until no more rules can be applied. If the simplified terms *reduce(cG)* and *reduce(cR)* are found to be equal, the process is repeated at the next comparison point until all comparison points have been proven equal.

3.1 Verilog to TRS Translation (translate())

Our approach begins with the translation of the source Verilog modules into a *structural* TRS that "simulates" the Verilog evaluation semantics, that is, it exactly follows the behavior and evaluation semantics of the Verilog design. As such, the structural TRS is a syntactically translated entity that is at the same level of abstraction as the Verilog design. The result of this translation is a rewrite system that can be used to compute the symbolic term for any signal in terms of other signals and/or primary inputs. Each hierarchical signal is represented by a new constant function symbol (*signal function*). Therefore, the hierarchy is "flattened" in this structural TRS. Rewrite rules rewrite each signal function into an expression consisting of RTL operators and other signal functions. Consider the example translation of the following Verilog into a structural TRS:

```
module top(inA, inB, opt, sel, out);
  input inA, inB, opt, sel;
  output out;
  reg out;
  wire fout;
  foo f1 (fout, inA, inB);
  always@* begin
    out = sel ? fout : opt;
    out = inA | out;
  end
endmodule
module foo (S, A, B);
  input A, B;
  output S;
  wire S;
  assign S = A ∧ B;
endmodule
```

The resulting structural TRS for the module `top` is the following:

$$\texttt{f1.A}() \rightarrow \texttt{inA}()$$
$$\texttt{f1.B}() \rightarrow \texttt{inB}()$$
$$\texttt{f1.S}() \rightarrow (\texttt{f1.A}() \wedge \texttt{f1.B}())$$
$$\texttt{fout}() \rightarrow \texttt{f1.S}()$$
$$\texttt{out1}() \rightarrow \texttt{if}(\texttt{sel}(), \texttt{fout}(), \texttt{opt}())$$
$$\texttt{out2}() \rightarrow (\texttt{inA}() \mid \texttt{out1}())$$

Since the signal `out` is assigned more than once, the different assignments are split into two signal functions, `out1` and `out2`. We assume, for simplicity in the Verilog modules, that primary inputs are never assigned and primary outputs are never referenced.

We assume that the input Verilog is race free (that is, no multiple parallel assignments for the same signal) and loop free (that is, no cyclic dependencies between combinational `always` blocks). The resulting structural TRS will then be convergent, that is, confluent (due to the race-free assumption) and terminating (due to the loop-free assumption). The loop-free and race-free assumptions can be checked by standard Verilog linting tools. Note that, for this structural TRS, the Verilog RTL operators are uninterpreted; the structural TRS is only used to construct terms defining the values of signals in terms of other signals.

3.2 Computing Comparison Points (computeComparePoints())

The *computeComparePoints()* function finds the intermediate comparison points using a heuristic. We have mentioned that the structural TRS can be used to find the symbolic term for any signal in the design. If all of the bits of the signal are assigned together (in the same rewrite step), there will be a single corresponding symbolic term for that signal. However, if the bits of the signal are assigned separately (different rewrite steps), there will be more than one symbolic term for the signal. We clarify this with an example.

1. We use the word *signal* to refer to Verilog variables in RTL modules and reserve *variable* for variables in TRSs.

Consider a 32-bit multiplier that we would like to verify, which has `mul_result[31:0]` as an output signal. If the multiplier RTL has only one assignment statement assigning the entire value `mul_result[31:0]`, then there is only one rewrite step that results in generating the symbolic term for `mul_result`. Therefore, there will be a single symbolic term for `mul_result`.

Assume that the multiplier's RTL description is written such that 8 bits of the output are assigned a value together, that is, at the same time. All of the 32 bits of the output are therefore assigned values after four such assignments. In the TRS for this multiplier, there will be four rewrite steps corresponding to these four assignments. Each rewrite step generates a symbolic term for `mul_result`. Hence, there will be four symbolic terms that correspond to `mul_result[7:0]`, `mul_result[15:8]`, `mul_result[23:16]`, and `mul_result[31:24]`.

Every subset of bits assigned will thus have its symbolic term. We will call such assignments (to different subsets of bits of the same signal) *reassignments* in the rest of the paper. Thus, a set of reassignments for a signal define a partition of the bits for the signal. In our example, the four *reassignments* define the partition `{[31:24], [23:16], [15:8], [7:0]}` on the 32 bits of the output signal `mul_result`.

Let the golden multiplier design be a shift-and-add design that has `golden_mul_result` as an output signal. The RTL description for this design assigns a value to the output 1 bit at a time. Therefore, in the TRS of the golden design, there will be 32 *reassignments* defining the partition $\{31, 30, \ldots 2, 1, 0\}$ on the bits of the signal `golden_mul_result`.

Comparison points are computed for every (declared) output signal in the two designs in the following way:

1. The reassignment bit partitions in the golden and revised designs are computed. In our example, the golden partition is $\{31, 30, \ldots 2, 1, 0\}$, and the revised partition is `{[31:24], [23:16], [15:8], [7:0]}`.
2. A new variable is defined for every set of bits in the pairwise intersection of these two partitions. In our example, the pairwise intersection groups entries of the golden partition together. The new variables in the golden design will be `mG1`, `mG2`, `mG3`, and `mG4`, corresponding to the bit sets $\{7, 6, \ldots 1, 0\}$, $\{15, 14, \ldots 9, 8\}$, $\{23, 22, \ldots 17, 16\}$, and $\{31, 30, \ldots 25, 24\}$, respectively. The new variables in the revised design will be `mR1`, `mR2`, `mR3`, and `mR4`, corresponding to the bit sets `{[31:24], [23:16], [15:8], [7:0]}`, respectively.
3. The new variables obtained are paired to define the set of comparison points. In our example, the set of comparison points is `{(mG1, mR1), (mG2, mR2), (mG3, mR3), (mG4, mR4)}`.

We thus compute a partition of the bits for a particular output defined by the reassignments in both the golden and revised designs. This is a simple heuristic that appears to work well for arithmetic circuits with common outputs and possibly some common internal points.

It is important to note that these comparison points are computed before the equivalence checking process by statically analyzing the two RTL descriptions. They do not form a part of the equivalence checking algorithm itself. These comparison points are used for decomposing the equivalence checking space to create smaller, more tractable problems to be proven by the equivalence checking technique, which, in our case, is term rewriting. The scalability of the algorithm therefore does not depend on the computation of comparison points.

Traditional combinational equivalence checkers routinely face the issue of not being able to reliably conclude that the two designs are not equivalent. This is the problem of *false negatives*. In order to mitigate the verification complexity, equivalence checkers perform a *hierarchical verification* that is comprised of isolated verification of each hierarchical block under the assumption that the exact functional equivalence at hierarchical boundaries is preserved. False negatives occur in such hierarchical verification when either 1) the functional equivalence at hierarchical boundaries is not preserved or 2) the block of the design is functionally equivalent only when it is constrained by the environment and not with unconstrained variables as viewed by the hierarchical verification process [1]. These issues are circumvented by our technique since we "flatten" the hierarchy during the translation of the design from Verilog to TRS, as shown in the example in Section 3.1, as well as the Ripple Carry Adder (RCA) and Carry Lookahead Adder (CLA) examples in Section 3.4. We verify the designs without maintaining their hierarchical boundaries and, therefore, do not encounter the false-negative problem in our technique.

Another noteworthy difference between our technique and traditional gate-level equivalence checkers is that we do not view each internal register as a comparison point. Our comparison points are the assignments or reassignments to the output signals of the design. Consequently, between arithmetic designs whose (output) size and number of outputs are equivalent, a correspondence between the comparison points in the two designs can always be expected.

3.3 Checking Equivalence of Terms (reduce())

The *reduce()* function checks for equivalence between two symbolic terms by rewriting based on simplification. This is achieved by using a separate database of rewrite rules that codify various identities about the RTL operators. For example, one may introduce an absorption and association rule for `&`, as well as rules for reducing arithmetic and left shifts:

$$(x \,\&\, x) \rightarrow x, \tag{1}$$

$$((x \,\&\, y)\&z) \rightarrow (x \,\&\, (y \,\&\, z)), \tag{2}$$

$$(x << 3) \rightarrow (x << 2) + (x << 1) + (x << 1), \tag{3}$$

$$((x << 1) - x) \rightarrow x, \tag{4}$$

$$((x << 1) << 1) \rightarrow (x << 2). \tag{5}$$

These rules can be generic or design specific, as demonstrated in Section 5.3. At any comparison point,

```
module rca16bit(A, B, Cin, S, Cout);
input [15:0] A, B;
input Cin;
output [15:0] S;
output Cout;
reg S, Cout;
wire [14:0] Carry;

rca1bit rca1bit0(A[0], B[0], Cin, S[0], Carry[0]);       R1
rca1bit rca1bit1(A[1], B[1], Carry[0], S[1], Carry[1]);  R2
                  ⋮
rca1bit rca1bit15(A[15], B[15], Carry[14], S[15], Cout); R16
endmodule

module rca1bit(A, B, C, S, Cout);
input A, B, C;
output S, Cout;
assign S = A ∧ B ∧ C;
assign Cout = A&B | B&C | C&A;
endmodule
```

Fig. 2. RCA Verilog.

while trying to prove equivalence, the *reduce()* function selects a set of rewrite rules from the database and applies them in some order. If the resulting simplifications fail to prove term equivalence, a different set (and/or ordering) of rewrite rules is chosen and applied. These *proof iterations* terminate when term equivalence is established or when no more rules can be applied.

When two designs are declared unequal by our technique, the result is reliable unless the rules in the rule base are not sufficient to simplify the terms under comparison. In that case, the prover part of our tool gives the proof trace at the last comparison point. User intervention is required to check the proof for more rules or to accept the negative result.

However, it can be inferred that two terms are equivalent if their simplified values are equal. We will discuss this term reduction function in a later section, but we note that the procedure is not complete in determining term equivalence. Instead, *reduce()* is designed to be efficient and sufficient for the domain of circuits to be analyzed. In cases where the set of rules used by *reduce()* is insufficient, we allow the user to augment this set. The decomposition of the equivalence check using comparison points and incremental refinement lessens the requirements on efficiency, as well as sufficiency for the function *reduce()*.

3.4 Example: Adder Verification

We present an illustrative example of how our technique works on adder circuits. We verify a 16-bit CLA design. We use a simple RCA as the golden design for adders. It adds two vectors by doing a bitwise xor and generates a corresponding carry bit. The Verilog code for a 16-bit RCA design is shown in Fig. 2.

The structural TRS is obtained by applying the *translate()* function to the RCA. The structural TRS for the module `rca16bit` is the following:

```
R1: rca1bit0.A()  → A[0]()
    rca1bit0.B()  → B[0]()
    rca1bit0.C()  → Cin()
    rca1bit0.S()  → (rca1bit0.A() ∧ rca1bit0.B() ∧
                     rca1bit0.C())
    rca1bit0.Cout() → ((rca1bit0.A() &
                        rca1bit0.B()) |
                       (rca1bit0.B() & rca1bit0.C()) |
                       (rca1bit0.C() & rca1bit0.A()))
    S[0]()        → rca1bit0.S()
    Carry[0]()    → rca1bit0.Cout()
R2: rca1bit1.A()  → A[1]()
    rca1bit1.B()  → B[1]()
    rca1bit1.C()  → Carry[0]()
    rca1bit1.S()  → (rca1bit1.A() ∧
                     rca1bit1.B() ∧ rca1bit1.C())
    rca1bit1.Cout() → ((rca1bit1.A() &
                        rca1bit1.B()) |
                       (rca1bit1.B() & rca1bit1.C()) |
                       (rca1bit1.C() & rca1bit1.A()))
    S[1]()        → rca1bit1.S()
    Carry[1]()    → rca1bit1.Cout()
       ⋮
R16: rca1bit15.A() → A[15]()
    rca1bit15.B()  → B[15]()
    rca1bit15.C()  → Carry[14]()
    rca1bit15.S()  → (rca1bit15.A() ∧
                      rca1bit15.B() ∧ rca1bit15.C())
    rca1bit15.Cout() → ((rca1bit15.A() &
                         rca1bit15.B()) |
                        (rca1bit15.B() & rca1bit15.C()) |
                        (rca1bit15.C() & rca1bit15.A()))
    S[15]()        → rca1bit15.S()
    Cout()         → rca1bit15.Cout()
```

We observe that labels **R1** ... **R16** in Fig. 2 correspond to the set of rules `R1` ... `R16` in the structural TRS.

The target design, a CLA, is similarly translated from its Verilog implementation to a structural TRS. The Verilog code for the CLA is given in Fig. 3.

There are four module calls to the `cla4bit` module by the main module. There are four `cla4bit` blocks in the design. The `cla4bit` module computes four successive carries at a time. The sum for the corresponding four bits is calculated in this module. The `cla4bit` module also calls the `PGgen` module, which generates the `Ps` (propagated carries) and the `Gs` (generated carries) for the block. A module called `fastcarry` is called to calculate the input carry values `(Cin, C[3], C[7], C[11])` for each of the four `cla4bit` blocks.

The structural TRS for the module `cla16bit` is the following (not complete):

```
R1: C3()  → fc.c3()
    C7()  → fc.c7()
    C11() → fc.c11()
       ⋮
R2: cla0.a[0]() → A[0]()
    cla0.b[0]() → B[0]()
    cla0.s[0]() → (cla0.a[0] ∧ cla0.b[0] ∧ cla0.c[0])
    cla0.a[1]() → A[1]()
    cla0.b[1]() → B[1]()
    cla0.s[1]() → (cla0.a[1] ∧ cla0.b[1] ∧ cla0.c[1])
    cla0.a[2]() → A[2]()
    cla0.b[2]() → B[2]()
```

```
module cla16bit (A, B, Cin, S, Cout);
 input [15:0] A, B;
 input Cin;
 output [15:0] S;
 output Cout;
 reg S, Cout;
 wire C3, C7, C11;
 fastcarry fc (A, B, Cin, C3, C7, C11);     R1
 cla4bit cla0 (A[3:0], B[3:0],Cin, S[3:0]); R2
 cla4bit cla1 (A[7:4], B[7:4], C3, S[7:4]); R3
 cla4bit cla2 (A[11:8],B[11:8],C7,S[11:8]); R4
 cla4bit cla3 (A[15:12], B[15:12], C11, S[15:12]); R5
endmodule

module cla4bit (a, b, cin, s);
 input [3:0] a, b;
 input cin;
 output [3:0] s;
 wire [3:0] c;
 assign c[0] = g[0] | p[0]&cin;
 assign c[1] = g[1] | g[0]&p[1] | p[1]&p[0]&cin;
 assign c[2] = g[2] | g[1]&p[2] | g[0]&p[2]&p[1] | p[2]&p[1]&p[0]&cin;
 assign c[3] = g[3] | g[2]&p[3] | g[1]&p[3]&p[2]
          | g[2]&p[3]&p[2]&p[1] | p[3]&p[2]&p[1]&p[0]&cin;
 assign s[0] = a[0] ∧ b[0] ∧ c[0];
 assign s[1] = a[1] ∧ b[1] ∧ c[1];
 assign s[2] = a[2] ∧ b[2] ∧ c[2];
 assign s[3] = a[3] ∧ b[3] ∧ c[3];
 PGgen pg0 (a[0], b[0], p[0], g[0]);
 PGgen pg1 (a[1], b[1], p[1], g[1]);
 PGgen pg2 (a[2], b[2], p[2], g[2]);
 PGgen pg3 (a[3], b[3], p[3], g[3]);
endmodule

module PGgen (a, b, p, g);
 input a, b;
 output p, g;
 assign p = a ∧ b;
 assign g = a & b;
endmodule

module fastcarry (a, b, cin, c3, c7, c11);
/*Accelerated Carry Computation -- not detailed.*/
endmodule
```

Fig. 3. CLA Verilog.

```
   cla0.s[2]() → (cla0.a[2] ∧ cla0.b[2] ∧ cla0.c[2])
   cla0.a[3]() → A[3]()
   cla0.b[3]() → B[3]()
   cla0.s[3]() → (cla0.a[3] ∧ cla0.b[3] ∧ cla0.c[3])
   cla0.cin()  → Cin()
   S[3:0]()    → cla0.s[3](), cla0.s[2],
                 cla0.s[1](), cla0.s[0]
      ⋮
R5:cla3.a[0]() → A[12]()
   cla3.b[0]() → B[12]()
   cla3.s[0]() → (cla3.a[0] ∧ cla3.b[0] ∧ cla3.c[0])
   cla3.a[1]() → A[13]()
   cla3.b[1]() → B[13]()
   cla3.s[1]() → (cla3.a[1] ∧ cla3.b[1] ∧ cla3.c[1])
   cla3.a[2]() → A[14]()
   cla3.b[2]() → B[14]()
   cla3.s[2]() → (cla3.a[2] ∧ cla3.b[2] ∧ cla3.c[2])
   cla3.a[3]() → A[15]()
   cla3.b[3]() → B[15]()
   cla3.s[3]() → (cla3.a[3] ∧ cla3.b[3] ∧ cla3.c[3])
   cla3.cin()  → Cin()
   S[15:12]()  → cla0.s[3](), cla0.s[2],
                 cla0.s[1](), cla0.s[0]
```

Labels **R1** ... **R5** in Fig. 3 correspond to the sets of rules R1 ... R5 in the structural TRS.

We are interested in showing equivalence with respect to the primary output signals S and Cout. The *computeComparePoints()* function now calculates the comparison points. The signals S and Cout are reassigned in the design. If the same subset of bits is reassigned in both designs, it is recognized as a comparison point. In the RCA, every rewriting step assigns a value to a single bit of the output signals. For instance, R1 in the RCA represents a single rewrite step that assigns a value to S[0]. However, in the CLA, every rewriting step (that is, R2 ... R5 in the TRS for CLA) assigns a value to four bits of the sum (that is,

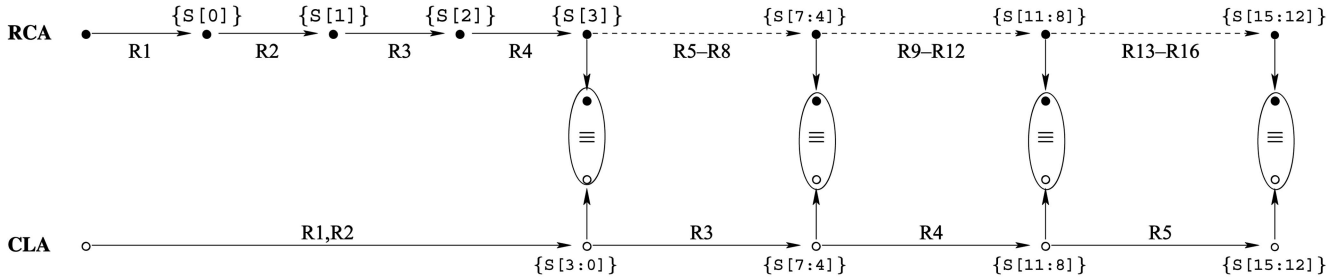


Fig. 4. Proof of correctness of the CLA compared against the RCA. • represents terms of the RCA. ○ represents terms of the CLA. The variable within {} is the reassigned output. ≡ represents the equivalence between the outputs at each comparison point.

S[3 : 0] . . . S[15 : 12]). Therefore, a single step in CLA corresponds to four steps in the RCA. A comparison point is recognized at S[3]. The symbolic terms obtained in both designs for S[3] are compared.

The *reduce()* function proves equivalence at the first comparison point. It is simple to understand how the symbolic terms generated are equivalent by tracing the rewrite steps in both of the TRSs that lead to the (reassigned) outputs. A correspondence between the rules for the two TRSs is given below. Fig. 4 explains this in an intuitive manner.

Applying rules R1 . . . R4 in the RCA TRS corresponds to rule R2 in the CLA TRS since the four bits of the sum are computed as an xor of the corresponding input operand and carry bits. However, the input carry terms in the two TRSs are different. The symbolic term for Carry[3] in the RCA is obtained by applying rules R1 . . . R4. The corresponding term in the CLA TRS, c[3], is obtained from rule R2. Applying rule R2 of the CLA TRS initially gives this fourth-stage carry in terms of the Ps and Gs of the previous stages and subsequently gives the same expression in terms of A and B instead of Ps and Gs. Therefore, the symbolic values of the carry terms in both of the TRSs turn out to be exactly equal.

Once S[3] is verified, the same procedure is repeated at the remaining comparison points, namely, S[7], S[11], and S[15]. The normal form of both of the TRSs is reached when S[15] is computed. The two TRSs are thereby proven equivalent.

3.5 Verifire: An Automated Proof Generator

Our tool, Verifire, automates the algorithm introduced in Section 3. Vtranslate automates the *translate()* function, and Vprove automates the *prove()* function of the algorithm.

Vtranslate is a compiler that accepts Verilog (synthesizable by commercial tools) as input. It automatically identifies and flattens the module hierarchy and constructs the structural TRS for the entire circuit. Vprove automatically generates equivalence proofs between two TRSs using the technique of stepwise refinement, outlined in Section 3. The golden TRS and the revised TRS are inputs to the proof engine. The tool automatically generates a proof or returns an error trace if it cannot establish the proof.

Vprove implements the *computeComparePoints()* and *reduce()* functions. The *reduce()* function is called as and when each comparison point is computed. The tool keeps track of potential multiple reduction rules and implements a backtracking algorithm in order to establish the equivalence at each comparison point.

Verifire was implemented in C++ and was used to prove many multiplier circuits. The tool can automatically generate proofs for standard multiplier designs like the *Booth* multiplier, *Array* multipliers, and *Tree* multipliers. It can also automatically generate proofs for multiplier designs that are modifications of these standard designs.

4 TRS Equivalence

This section provides a technical definition of TRS equivalence and proof that the decomposition of TRS equivalence we use (via comparison points) is sound. This section is not requisite to the main points of this paper and could be skipped by the reader if desired.

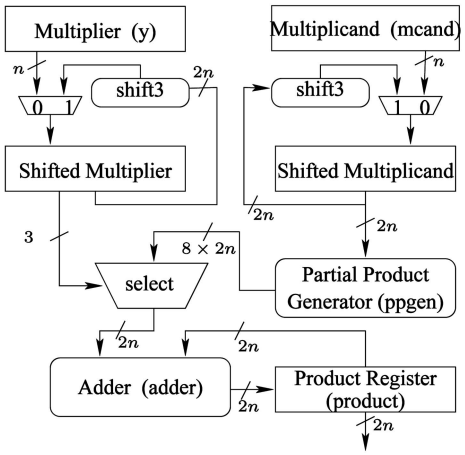
Given a specified top module M of a Verilog design, the *primary inputs* $PI(M)$ are the signal functions for the primary input signals of M and the primary outputs $PO(M)$ are the corresponding signal functions for the primary output signals.

Using the structural TRS for top module M, we can define the function $\Phi_M(s, X)$, which takes a signal function s and a set of signal functions X and returns the normal form of s in the rewrite system R minus the rules for rewriting functions in $X \setminus \{t\}$.

We need to define the notion of equivalence that we wish to check. For a term t, we define the $support(t)$ to be the set of signal functions and variables in t. For two terms s and t, we define $s \equiv t$ as $support(s) = support(t)$ and, for all ground substitutions σ of $X = support(s)$, we have $s[X/\sigma] = t[X/\sigma]$.[2] Given modules G and R (the golden and revised modules where $PI = PI(G) = PI(R)$ and $PO = PO(G) = PO(R)$) and a set of signal functions X, define $R \sim_X G$ as $(\forall t \in X : \Phi_G(t, X) \equiv \Phi_R(t, X))$. Note that, in order for this definition to be useful, we assume that the *equivalent* signals in the two modules, G and R, have the same signal name. In cases of reassignments to the same signal in either module, we may need to adjust the names assigned to the reassigned signals in order to ensure correspondence.

We wish to prove $R \sim_{PO} G$. As explained earlier, we decompose the proof by computing a set of *comparison point*

2. We use the notation $t[X/\sigma]$ to denote the term t, where variables in X have been replaced according to the assignment σ. If σ does not assign to any variables in set X, then the term t is unchanged.



multiplier bits	partial product generated x : multiplicand
0 0 0	pp0: 0
0 0 1	pp1: x
0 1 0	pp2: $x << 1$
0 1 1	pp3: $x << 1 + x$
1 0 0	pp4: $x << 2$
1 0 1	pp5: $x << 2 + x$
1 1 0	pp6: $x << 2 + x << 1$
1 1 1	pp7: $x << 3 - x$

(b)

Fig. 5. (a) Architecture of a Booth multiplier. (b) Partial product terms of the Booth multiplier.

signal functions C. We make use of the following property to transfer the result to the proof of equivalence for PO:

Theorem 1.

$$\forall R, G, C, O : (((R \sim_C G) \wedge (O \subseteq C)) \Rightarrow (R \sim_O G)).$$

Proof outline. Take an arbitrary signal function $s \in O$. We first observe that the term $\Phi_R(s, O)$ is equal to the iterative expansion beginning with the term $\Phi_R(s, C)$, where, in each step, the signals $x \in C \setminus O$ are substituted by the corresponding terms $\Phi_R(x, C)$. A similar observation holds for $\Phi_G(s, O)$. Then, for any ground substitution for O, one can prove by induction, following the iterative expansions we observed, that the desired equality holds between $\Phi_R(s, O)$ and $\Phi_G(s, O)$ using the assumption $R \sim_C G$ to relieve the induction hypothesis and substituting equals for equals along the way. □

Thus, we prove $R \sim_{PO} G$ by proving $R \sim_C G$ instead, where $PO \subseteq C$. The following additional (trivial) property is useful in chaining $\sim$ proofs together:

Theorem 2.

$$\forall R, I, G : (((R \sim_{PO} I) \wedge (I \sim_{PO} G)) \Rightarrow (R \sim_{PO} G)).$$

Our procedure for proving $R \sim_{PO} G$ consists of the following two steps: Compute a set of *comparison points* C and then prove $R \sim_C G$.

In order to check $R \sim_C G$, we iterate through each signal $x \in C$, compute $s = \Phi_R(x, C)$ and $t = \Phi_G(x, C)$, and check if $s \equiv t$. The mechanism for checking $s \equiv t$ for two given terms is the simplification function $reduce(t)$, which maps a term t to a reduced term that is equal under all substitutions to t. If $reduce(s) = reduce(t)$, we can deduce that $s \equiv t$.

5 Multiplier Verification

We consider the space of multipliers divided into *standard* and *nonstandard* multipliers. The standard multipliers are the widely used common multiplier designs like Booth, Wallace tree, Dadda Tree, and Array multipliers. The nonstandard multipliers have incremental optimizations made to these standard multiplier designs. We have extended our technique to cover the space of these two categories of multipliers. We illustrate our technique on the Booth multiplier and BISMUL, an optimization of the Booth multiplier.

5.1 Booth Multiplier

The multiplier in Fig. 5a is a 64-bit radix-3 nonoverlapping Booth multiplier. The `ppgen` block generates the eight partial products that form the Booth encoding. The `ppsel` block selects the relevant partial product depending on the incoming bits from the multiplier. The partial products are added in the adder block. The `ppgen` is given the shifted multiplicand as input (`shift3`) to generate the current partial product. This method is repeated for all of the bits of the multiplier. The result appears in the `product` register.

To prove the functional correctness of the above design, we follow the technique explained in Section 3. We illustrate the proof using the outline of the proof provided in that section.

We use a simple shift-and-add multiplier as the reference TRS for multipliers. It performs multiplication by generating partial products. It shifts the multiplicand to the left by one bit after every partial product calculation. The partial product of the current stage is set to the sum of the previous partial product and the shifted multiplicand of the current stage or 0, depending on whether the multiplier bit corresponding to the current stage is 1 or 0. The Verilog code of the shift and add calls a `shift` and an `add` module iteratively.

The target design here is the Booth multiplier discussed above. Vtranslate extracts its TRS from the Verilog code.

In the case of the Booth multiplier, the PO needed to prove $(R \sim_{PO} G)$ is `product`, as explained in Section 3. A sketch of this proof (as output by the tool) follows.

The first comparison point in the proof is after 3 bits of output (`product`) are updated in both TRSs. This is because the Booth updates 3 bits of its `product` simultaneously as opposed to shift and add, which updates its `product` sequentially. The output of the tool after the first

comparison point is given as follows, where Stage i in the tool output represents the ith update of product:[3]

```
Comparison Point 1:
Reference Model: Shift-and-Add
Stage 1.
Rule x : product = product + mcand          if(y[0])
Rule y : product = product + 0              if(y[0])
Stage 2.
Rule x : product = product + mcand << 1     if(y[1])
Rule y : product = product + 0              if(y[1])
Stage 3.
Rule x : product = product + mcand << 2     if(y[2])
Rule y : product = product + 0              if(y[2])
Revised Model: Booth
Stage 1.
Rule a : product = product + 0           if (y[0] &
                                         y[1] & y[2])
Rule b : product = product + mcand       if (y[0] &
                                         y[1] & y[2])
Rule c : product = product + mcand << 1  if (y[0] &
                                         y[1] & y[2])
Rule d : product = product + mcand << 1 + mcand
                           if (y[0] & y[1] & y[2])
Rule e : product = product + mcand << 2  if (y[0] &
                                         y[1] & y[2])
Rule f : product = product + mcand << 2 + mcand
                           if (y[0] & y[1] & y[2])
Rule g : product = product + mcand << 2 + mcand << 1
                           if (y[0] & y[1] & y[2])
Rule h : product = product + mcand << 3 - mcand
                           if (y[0] & y[1] & y[2])
```

The expressions generated from both of the TRSs from the first comparison point are displayed with their corresponding rules. For instance, Reference.Stage 1.Rule x is product = product + mcand if (y[0]).

Correspondence at the comparison point is established by a case-by-case analysis of the rules. Every encoding of the Booth multiplier is compared to the shift and add, with the same conditions. For instance, Revised.Stage 1.Rule a gives the expression for the partial product generated for the Booth encoding 000. Applying the condition y[0]y[1]y[2] = 000 on the shift and add corresponds to rules Reference.Rule 1y, Reference.Rule 2y, and Reference.Rule 3y. Such an analysis is performed for all cases.

The *reduce()* function is applied by Vprove to simplify corresponding terms at Comparison Point 1. An outline of this simplification as output by the tool is given below:

```
Correspondence (after case analysis):
Revised          Reference
Rule 1a == Rule 1y, Rule 2y, Rule 3y
Rule 1b == Rule 1x, Rule 2y, Rule 3y
Rule 1c == Rule 1y, Rule 2x, Rule 3y
Rule 1d == Rule 1x, Rule 2x, Rule 3y
Rule 1e == Rule 1y, Rule 2y, Rule 3x
Rule 1f == Rule 1x, Rule 2y, Rule 3x
Rule 1g == Rule 1y, Rule 2x, Rule 3x
Rule 1h == Rule 1x, Rule 2x, Rule 3x
```

The set of rules used by *reduce()* is not complete, which renders establishing this correspondence nontrivial. In many situations, the user might need to add to this set of rules to assist Vprove in establishing the correspondence. We illustrate this with the example of Rule 1h == Rule 1x, Rule 2x, Rule 3x.

Revised.Rule 1h under the condition y[0]y[1]y[2] = 111 states that

$$\texttt{product} \rightarrow \texttt{product} + \texttt{mcand} << 3 - \texttt{mcand}.$$

The corresponding rules in the reference model induced by the condition are Reference.Rule 1x, Reference.Rule 2x, and Reference.Rule 3x. Combining the three reference rules, we obtain the term

$$\texttt{product} + \texttt{mcand} + \texttt{mcand} << 1 + \texttt{mcand} << 2.$$

Applying *reduce()* on the revised term, we obtain the term

$$\texttt{product} + (\texttt{mcand} << 2 + \texttt{mcand} << 1 + \texttt{mcand} << 1) - \texttt{m}.$$

However, if the database has no further rules to invoke *reduce()*, then we cannot establish the correspondence. Hence, the user needs to add a rule to the database to help simplify the term further. In this case, we add the rule

$$((\texttt{x} + \texttt{y}) - \texttt{z}) \rightarrow (\texttt{x} + (\texttt{y} - \texttt{z})).$$

With the addition of this rule, the revised term now is reduced to

$$\texttt{product} + (\texttt{mcand} << 2 + \texttt{mcand} << 1) + (\texttt{mcand} << 1 - \texttt{m})$$

which is further reduced to

$$\texttt{product} + \texttt{mcand} << 2 + \texttt{mcand} << 1 + \texttt{mcand}.$$

Thus, we establish the correspondence.

The proof now proceeds to subsequent comparison points iteratively till the output is obtained in its normal form. Therefore, $R \sim_C G$ is proven. Fig. 6 explains the proof in an intuitive manner.

5.2 BISMUL

In order to improve the performance of multipliers, more complicated algorithms and designs are used. We consider a high-performance multiplier, BISMUL [38], that is a modification of the Booth multiplier. A radix-3 Booth multiplier architecture using 3-bit scan with no overlap invariably generates dummy bits in the last 3-bit scan. In BISMUL, this last 3-bit scan is moved to the first 3-bit scan so that the dummy bits can be used for odd multiple generation. The sequence for bit scanning is shown in Fig. 8. The improvement in the multiplication speed in BISMUL is obtained by executing several Partial Product Selectors (PPSELs) in parallel. The shifting sequence of the multiplier decides the inputs to the PPSELs. These selected partial products are summed through carry-save additions.

The architecture of a 64×64 bit multiplier is shown in Fig. 7. This architecture is comprised of Product Registers

3. The rules in the output are reproduced in pseudo-Verilog syntax and they correspond to the rewrite rules of the TRS, as described in Section 3. For example, rules Reference.Stage 1.Rule x and Reference.Stage 1.Rule y in the TRS would be the rewrite rule product() → product() + if (y[0](), mcand(), 0).

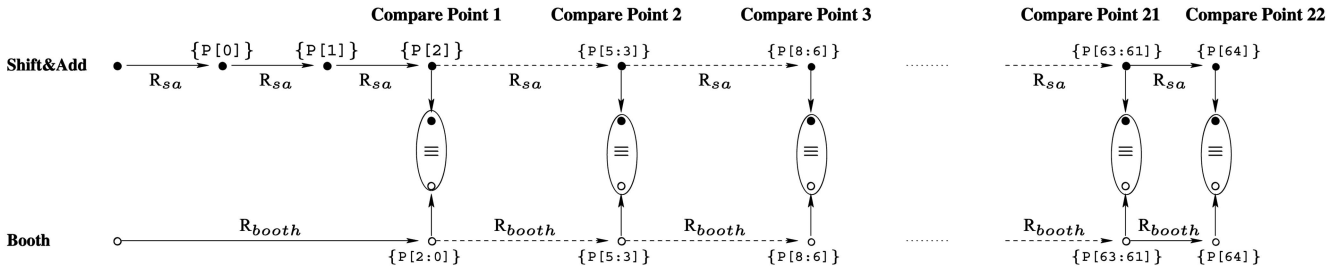


Fig. 6. Proof of correctness of the Booth multiplier compared to the shift-and-add multiplier. • represents terms of the shift-and-add multiplier. ○ represents terms of the Booth multiplier. R_{sa} represents the rules of the shift-and-add multiplier (`Rule x` and `Rule y`). R_{booth} represents the corresponding Booth multiplier rules (`Rule a...Rule h`). The variable `product` is the reassigned output. ≡ represents the equivalence between the outputs at each comparison point.

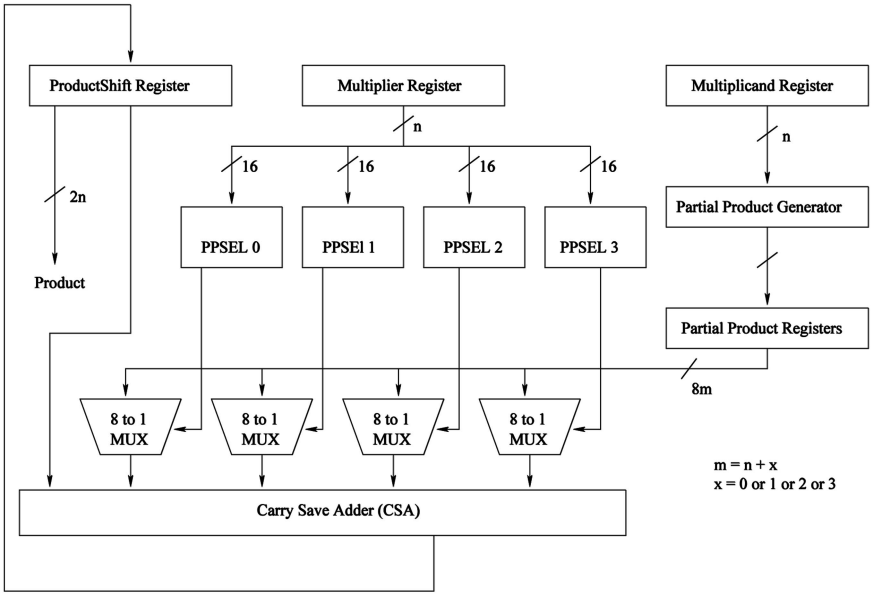


Fig. 7. Architecture of a BISMUL.

(PRs), Partial Product Generators (PPGs), PPSELs, Multiplexers, and a Carry-Save Adder (CSA). PPG is implemented according to Fig. 8. There are four PPSELs. Each PPSEL has 16-bit inputs, whose sequence is shown in Fig. 8. The operation of the BISMUL is as follows: In the first cycle, PPG generates eight partial products and each PPSEL selects one partial product. The two dummy bits in the lower bit position in the first three bits of PPSEL cause the selection of either 0 or four times the multiplicand (000 or 100). The partial products are added and stored in the PR. The partial products get generated in the first cycle. Subsequent cycles perform the same operation as described.

bit	Generation of Partial Product Terms
0 0 0	P0: 0
0 0 1	P1: multiplicand
0 1 0	P2: shift multiplicand left by 1
0 1 1	P3: add P1 and P2
1 0 0	P4: shift multiplicand left by 2
1 0 1	P5: add P1 and P4
1 1 0	P6: shift P3 to the left by one
1 1 1	P7: subtract P1 from 8

PPSEL	Inputs
PPSEL0	Multiplier[54:52],[42:40],[30:28],[18:16],[6:4][0]
PPSEL1	Multiplier[57:55],[45:43],[33:31],[21:19],[9:7][1]
PPSEL2	Multiplier[60:58],[48:46],[36:34],[24:22],[12:10][2]
PPSEL3	Multiplier[63:61],[51:49],[39:37],[27:25],[15:13][3]

Fig. 8. The partial product terms in a BISMUL and the inputs of each PPSEL.

We prove that the BISMUL is correct by using the following technique: We perform a series of reductions on the BISMUL to reduce it to its standard design, the Booth multiplier. The standard Booth multiplier is already verified using the above technique. Hence, the given nonstandard design can be proven correct.

Verifire extracts the corresponding TRSs from the BISMUL and Booth Verilog codes. The tool compares the modules in the nonstandard design (which derive from the modules in the standard design) to the corresponding modules in the standard design. The correspondence (and equivalence) between these "derived" modules of the nonstandard design and the modules in the standard design is established by the same method as described in Section 5.1 between the Booth and the shift-and-add.

In order to prove the reduction of BISMUL to Booth, it is enough to prove that the changed (terms) modules in BISMUL are equivalent to the original Booth terms. In this case, the terms `{ppsel0, ppsel1, ppsel2, ppsel3, mux8to1}` of the BISMUL form the revised design. The

Booth Multiplier	Verifire	Commercial Tool 1	Commercial Tool 2
$4b \times 4b$	16s	12s	9s
$8b \times 8b$	19s	20s	16s
$16b \times 16b$	24s	not completed	not completed
$32b \times 32b$	37s	not completed	not completed
$64b \times 64b$	53s	-	-
$128b \times 128b$	93s	-	-

(a)

Wallace Multiplier	Verifire	Commercial Tool 1	Commercial Tool 2
$4b \times 4b$	14s	10s	9s
$8b \times 8b$	18s	18s	16s
$16b \times 16b$	25s	not completed	not completed
$32b \times 32b$	40s	not completed	not completed
$64b \times 64b$	60s	-	-

(b)

Dadda Tree Multiplier	Verifire	Commercial Tool 1	Commercial Tool 2
$4b \times 4b$	13s	11s	8s
$8b \times 8b$	17s	19s	17s
$16b \times 16b$	29s	not completed	not completed
$32b \times 32b$	51s	not completed	not completed
$64b \times 64b$	83s	-	-

(c)

Fig. 9. Comparison of execution times of Verifire against two commercial equivalence checkers for (a) Booth, (b) Wallace Tree, and (c) Dadda Tree multipliers of varying sizes. In each case, the golden model was a shift-and-add multiplier of the corresponding size.

terms `{ppsel, shift3}` of the Booth act as the corresponding reference design. Similarly, the terms `{product-shift, carrysaveadder}` of BISMUL correspond to the `{ppsel, adder}` terms of Booth. Therefore, it is sufficient to prove the validity of each correspondence.

We have verified the BISMUL using our technique. We have also verified the Wallace Tree multiplier. The Booth multiplier we used was designed as a Booth-recoded Array multiplier. We have also verified a Dadda Tree multiplier using our technique. For each of these, we can also verify some modifications of the standard designs.

The terms in the TRS for the modified design are simplified to terms in the TRS for the standard design. The simplification is performed using the database of rules in Vprove. This set of rules is not exhaustive and may require manual intervention when presented with an entirely new design that does not build on the standard ones. However, for a large space of designs, it is completely automated.

Another aspect of our technique is the generality of the rules. Consider the case of a completely new design where our current set of rules is not sufficient to prove the equivalence. Following the proof trace, it is fairly straightforward to add the required rules for a 4-bit or 8-bit version of the multiplier. However, if the multiplier RTLs are modular, then these rules are general enough that the harder cases of 32-bit and 64-bit multiplier equivalence proofs are now completely automated.

Rule class	Number of rules	Example
Boolean	32	`(x | (y & z)) → ((x | y) & (x | z))`
Add/Subtract	44	`(x + (y - z)) → ((x + y) - z)`
Shift	16	`((x << 1) << 1) → (x << 2)`
Multiplier Specific	9	`((x << 1) - x) → x`
Total	101	

Fig. 10. Distribution of rewrite rules for multipliers used by the *reduce()* function of Vprove.

5.3 Results

We present the experimental results that we have obtained from our tool. We produce three sets of results on a radix-3 Booth multiplier, on a Wallace Tree multiplier, and on a Dadda Tree multiplier. The Booth multiplier is an array-based multiplier, whereas the Wallace multiplier has a tree of CSAs and a single CLA as the last stage. The Dadda Tree multiplier uses the more regular redundant binary addition trees [36] instead of a tree of CSAs. We show the time taken by the tool for increasing sizes of these multipliers.

We have tried to compare our tool to state-of-the-art equivalence checkers. Since equivalence checkers are most efficient when comparing two gate-level designs, we provided gate-level implementations of the Booth and Wallace Tree designs as inputs. Although our tool works at the RTL, we have compared the numbers obtained from the gate-level verification by the equivalence checkers with our tool output in order to provide a basis for comparison. It is seen in Figs. 9a, 9b, and 9c that the verification of 8×8 multipliers are performed by both Commercial Equivalence Checker 1 and Commercial Equivalence Checker 2 in a time comparable to our tool. However, in the case of 16×16 multipliers, both of the equivalence checkers do not run to completion. Our tool, in comparison, verifies the design in 24 sec. It can also be seen that, as the sizes increase, the time taken by our tool scales linearly with the size of the design.

The *reduce()* function uses a database of rewrite rules to prove equivalence. Fig. 10 illustrates a broad classification of the multiplier rewrite rules. We have classified the rules into those involving Boolean operators, add/subtract operators, and shift operators. These rules are mostly generic in nature and will form a part of the rule database for all multipliers. However, the rules classified into multiplier-specific rules are more specific to the design of the multiplier being verified. As mentioned earlier, the set of rewrite rules is not exhaustive. However, for the widely used multiplier designs, the necessary set of rewrite rules exists in the database, thereby making the rewriting very efficient.

We mentioned in Section 3.3 that the *reduce()* function proves term equivalence using proof iterations. Fig. 11 shows the number of proof iterations that the tool required to prove the equivalence of the 64×64 multipliers at two sample comparison points. The proof iterations for the Booth multiplier at (sample) comparison points 3 and 21 and the Wallace Tree multiplier at comparison points 3 and 7 have been illustrated. We observe that the proof iterations do not increase significantly as the proof progresses, that is, with increasing comparison points. We also observe that the number of comparison points for the Wallace Tree is less than that for the Booth. This is because the tree of CSAs in

Multiplier	Compare point	Number of rules	Number of proof iterations
Booth	3	107	192
Booth	21	107	212
Wallace Tree	3	107	347
Wallace Tree	7	107	291
Dadda Tree	3	107	462
Dadda Tree	7	107	341

Fig. 11. Number of proof iterations done by *reduce()* to prove equivalence at the given comparison points. The numbers correspond to the 64×64 Booth, Wallace Tree, and Dadda Tree multiplier designs.

the Wallace Tree design delays assignment to the final output bits. Therefore, the terms are larger and the effect of this is seen in the greater number of proof iterations for the Wallace Tree than that for the Booth multiplier.

In order to assist Commercial Equivalence Checker 1 in comparing RTL designs, we tried providing comparison points in the multiplier designs. These intermediary cut points were the partial products obtained in the two multipliers. The results of this experiment are displayed in Fig. 12. The Commercial Equivalence Checker runs to completion when assisted manually with comparison points for the 16×16 case. However, it fails to run to completion even when assisted by these comparison points when comparing two 32×32 bit or higher order multipliers. It may be noted that we have provided manual assistance with respect to the comparison points to the equivalence checkers, as opposed to our tool that generates these comparison points automatically. Our tool is effective in verifying multiplier designs that are modifications (usually for optimization) of standard multipliers like Booth, Wallace Tree, and Array multipliers. We used the tool for verifying the Verilog implementation of BISMUL [38], a complicated modified Booth multiplier. In this case, a Booth multiplier verified by our technique was used as the golden design and the BISMUL was the target design to be verified. Our tool caught a bug in the Verilog code that appeared while the tool tried to calculate the partial products after the first matching point. The expressions that the observed output (product) variable (`P`) was rewritten into could not be proven equal at the next matching point by Vprove. The rule correspondence that the tool had established, as well as the previous matching point, provided an error trace.

6 Conclusions

We have presented a new approach that uses stepwise refinement of TRSs for formally verifying arithmetic circuit designs. Equivalence checkers that use BDD-based algorithms [29] cannot handle large sizes of multipliers. Our tool manages to gracefully scale to large complex multipliers.

We provide a brief intuition as to why we demonstrate better performance. The first reason is due to our strategic decomposition of the equivalence checking state space. The process of computing comparison points once, before starting the verification process, by statically analyzing the RTL descriptions helps in decomposing the intractable problem. These comparison points help us obtain smaller equivalence checking problems, which can be verified using term rewriting or other equivalence checking engines. Since this process is done statically, it does not depend on the size of the design.

Another reason is because we represent circuits at a higher *term* level as opposed to the Boolean level representation used by the BDD-based techniques. The term representation we use is intuitive and easy. It is also more natural since the terms encapsulate the structural, as well as the functional, details of the circuit in a modular fashion. The manipulation of terms is also more efficient. Also, while comparing two TRSs, the problem of comparison can be reduced to a smaller problem since the two terms being compared need not be in their normal (canonical) form. However, BDDs are a canonical representation and their comparison cannot be easily decomposed. Our decomposition of two TRSs to pairs of comparable terms is also a significant reason for the efficiency of the technique.

The disadvantage in term rewriting is that the Vprove part of the tool, which implements the *reduce()* function introduced in Section 3, is incomplete. The *reduce()* function uses a database of rules to simplify the expressions it is comparing. This database of rules may require additional rules to simplify new expressions. However, this incompleteness is traded for the efficiency of the tool and, in practice, it can deal with many classes of large multipliers that were hitherto intractable with automated tools. We have incorporated a large number of rules that were needed to simplify the expressions that we encountered in the circuits we have targeted. In its current state, therefore, Vprove is very efficient for practical designs.

For arithmetic circuit verification, some techniques [8], [11], [12], [19], [37] are more effective than other model checking techniques. However, our technique achieves significantly more since we *automatically compute comparison points*. BMDs have been shown to verify 256-bit wide multipliers in reasonable time. However, BMDs are constructed by partitioning the circuit into components, with each component having simple word-level specifications. This custom crafting is not a feasible option for a generic design. Also, practical design optimizations tend to break

Multiplier	Verifire (Booth)	Commercial Tool (Booth)	Verifire (Wallace)	Commercial Tool (Wallace)	Verifire (Dadda)	Commercial Tool (Dadda)
$4b \times 4b$	16s	12s	14s	10s	13s	8s
$8b \times 8b$	19s	20s	18s	20s	17s	17s
$16b \times 16b$	24s	1942s	25s	972s	29s	not completed
$32b \times 32b$	37s	not completed	40s	not completed	51s	not completed
$64b \times 64b$	53s	-	60s	-	83s	-

Fig. 12. Comparison of execution times of Verifire against one commercial equivalence checker assisted by manual comparison points. Results are shown for Booth, Wallace Tree, and Dadda Tree multipliers.

such elegant component constructions. Our technique, on the other hand, does not need to construct artifacts according to the individual designs and is far more generic in scope. We are particularly able to handle complex design optimizations, as shown in Section 5.2. Verification times for BMDs are a quadratic function of the size of the multiplier, as opposed to our technique, where the times scale linearly with size. Additionally, BMDs are also prone to variable ordering issues (albeit to a lesser extent than BDDs) that require significant manual intervention. When the designs being checked are not found to be equivalent, the BMD algorithm may not terminate. In contrast, our tool aborts and provides an error trace for the bug instantly. Since equivalence checking is decomposed into comparison points, the exact location of the error can easily be detected at a failing comparison point.

Our technique is similar in spirit to a directed theorem-proving approach. However, our technique requires much less user expertise and ingenuity than theorem provers [14], [15], [17], [20], [23], [24], [28], [31], [33], [34]. Our tool is a dedicated arithmetic circuit checker and can be interfaced with equivalence checkers for arithmetic circuit verification. Another possibility is to integrate our tool with the theorem prover ACL2 [25] so that we can leverage the existing RTL library in ACL2 [16], [27] to add rules to Vprove in a sound manner.

Our technique is a step toward the verification of two generic arithmetic circuits. We have managed to verify a large number of arithmetic circuits using our technique, like adders, shifters, and comparators. This technique can tackle a large part of the multiplier space and many of the multipliers currently in use. We are also working on applying the technique to SRT division circuits and floating-point arithmetic circuits.

REFERENCES

[1] D. Anastasakis, L. McIlwain, and S. Pilarski, "Efficient Equivalence Checking with Partitions and Hierarchical Cut-Points," *Proc. 41st Ann. Conf. Design Automation (DAC '04),* pp. 539-542, 2004.

[2] S. Antoy and J. Gannon, "Using Term Rewriting to Verify Software," *IEEE Trans. Software Eng.,* vol. 20, no. 4, pp. 259-274, Apr. 1994.

[3] T. Arts and J. Giesl, "Applying Rewriting Techniques to the Verification of Erlang Processes," *Proc. 13th Int'l Workshop Computer Science Logic (CSL '99),* pp. 96-110, 1999.

[4] C.W. Barrett, D.L. Dill, and J.R. Levitt, "A Decision Procedure for Bit-Vector Arithmetic," *Proc. 35th Ann. Conf. Design Automation (DAC '98),* pp. 522-527, 1998.

[5] A.D. Booth, "A Signed Binary Multiplication Technique," *J. Mechanics and Applied Math.,* pp. 236-240, 1951.

[6] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers,* vol. 35, no. 8, pp. 677-691, Aug. 1986.

[7] R.E. Bryant, "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication," *IEEE Trans. Computers,* vol. 40, no. 2, pp. 205-213, Feb. 1991.

[8] R.E. Bryant and Y.-A. Chen, "Verification of Arithmetic Circuits with Binary Moment Diagrams," *Proc. 32nd Conf. Design Automation (DAC '95),* pp. 535-541, 1995.

[9] J.R. Burch, "Using BDDS to Verify Multipliers," *Proc. 28th Conf. ACM/IEEE Design Automation Conf.,* pp. 408-412, 1991.

[10] M.S. Chandrasekhar, J.P. Privitera, and K.W. Conradt, "Application of Term Rewriting Techniques to Hardware Design Verification," *Proc. 24th ACM/IEEE Design Automation Conf.,* pp. 277-282, 1987.

[11] J.-C. Chen and Y.-A. Chen, "Equivalence Checking of Integer Multipliers," *Proc. Asia and South Pacific Design Automation Conf. (ASP-DAC '01),* pp. 169-174, 2001.

[12] Y.-A. Chen and R.E. Bryant, "PHDD: An Efficient Graph Representation for Floating Point Circuit Verification," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design (ICCAD '97),* pp. 2-7, 1997.

[13] E.M. Clarke, M. Fujita, and X. Zhao, "Hybrid Decision Diagrams," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design (ICCAD '95),* pp. 159-163, 1995.

[14] D. Cyrluk, "Microprocessor Verification in PVS: A Methodology and Simple Example," Technical Report SRI-CSL-93-12, Menlo Park, Calif., 1993.

[15] D. Cyrluk, S. Rajan, N. Shankar, and M.K. Srivas, "Effective Theorem Proving for Hardware Verification," *Proc. Second Int'l Conf. Theorem Provers in Circuit Design, Theory, Practice, and Experience,* pp. 203-222, 1994.

[16] D.M. Russinoff, "A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD-K7 Floating-Point Multiplication, Division, and Square Root Instructions," *LMS J. Computation and Math.,* vol. 1, pp. 148-200, Dec. 1998.

[17] S.J. Garland, J.V. Guttag, and J.A. Staunstrup, "Verification of VLSI Circuits Using LP," *Proc. IFIP WG 10.2 Working Conf.: The Fusion of Hardware Design and Verification,* pp. 329-345, July 1988.

[18] T. Genet and F. Klay, "Rewriting for Cryptographic Protocol Verification," *Proc. 17th Int'l Conf. Automated Deduction (CADE '00),* pp. 271-290, 2000.

[19] H. Anderson, P. Williams, and H. Hulgaard, "Equivalence Checking of Combinational Circuits Using Boolean Expression Diagrams," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 18, no. 7, 1999.

[20] N. Harman, "Correctness and Verification of Hardware Systems Using Maude," technical report, Univ. of Wales, Swansea, 2000.

[21] J. Hoe and Arvind, "Hardware Synthesis from Term Rewriting Systems," *Proc. X IFIP Int'l Conf. VLSI (VLSI '99),* Nov. 1999.

[22] J.R. Burch, E.M. Clarke, D.E. Long, K.L. MacMillan, and D.L. Dill, "Symbolic Model Checking for Sequential Circuit Verification," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 13, no. 4, pp. 401-424, 1994.

[23] D. Kapur, "Theorem Proving Support for Hardware Verification," *Proc. Third Int'l Workshop First-Order Theorem Proving (FTP '00),* July 2000.

[24] D. Kapur and H. Zhang, "An Overview of Rewrite Rule Laboratory (RRL)," *J. Computer and Math. with Applications,* vol. 29, no. 2, pp. 91-114, 1995.

[25] M. Kaufmann and J. Moore, "ACL2: An Industrial Strength Version of NQTHM," *Proc. 11th Ann. Conf. Computer Assurance (COMPASS '96),* p. 23, 1996.

[26] J. Klop, "Term Rewriting Systems," *Handbook of Logic in Computer Science,* S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, eds., vol. 2, pp. 1-116. Oxford Univ. Press, 1992.

[27] M. Kaufmann and D. Russinoff, "Verification of Pipeline Circuits," *Proc. ACL2 Workshop 2000,* Technical Report TR-00-29, Dept. of Computer Sciences, Univ. of Texas at Austin, Oct. 2000.

[28] Z. Manna, N. Bjorner, A. Browne, E.Y. Chang, M. Colon, L. de Alfaro, H. Devarajan, A. Kapur, J. Lee, H. Sipma, and T.E. Uribe, "Step: The Stanford Temporal Prover," *Proc. Sixth Int'l Joint Conf. CAAP/FASE Theory and Practice of Software Development (TAPSOFT '95),* pp. 793-794, 1995.

[29] Y. Matsunaga, "An Efficient Equivalence Checker for Combinational Circuits," *Proc. 33rd Conf. Design Automation (DAC '96),* pp. 629-634, 1996.

[30] M. Mutz, "Using the HOL Prove Assistant for Proving the Correctness of Term Rewriting Rules Reducing Terms of Sequential Behavior," *Proc. Third Int'l Workshop Computer Aided Verification (CAV '91),* pp. 277-287, 1991.

[31] R. Boulton, A. Gordon, M.J.C. Gordon, J. Herbert, and J. van Tassel, "Experience with Embedding Hardware Description Languages in HOL," *Proc. Int'l Conf. Theorem Provers in Circuit Design: Theory, Practice, and Experience,* pp. 129-156, 1992.

[32] J. Sawada and W.A. Hunt, "Processor Verification with Precise Exceptions and Speculative Execution," *Proc. 10th Int'l Computer Aided Verification Conf. (CAV '98),* pp. 135-146, 1998.

[33] J.B. Saxe, S.J. Garland, J.V. Guttag, and J.J. Horning, "Using Transformations and Verification in Circuit Design," *Proc. Int'l Workshop Designing Correct Circuits,* J. Staunstrup and R. Sharp, eds., 1992.
[34] R. Sharp and O. Rasmussen, "Rewriting with Constraints in T-Ruby," *Proc. IFIP WG 10.5 Advanced Research Working Conf. Correct Hardware Design and Verification Methods,* pp. 226-241, 1993.
[35] X. Shen, "Design and Verification of Speculative Processors," *Proc. Workshop Formal Techniques for Hardware and Hardware-Like Systems,* June 1998.
[36] N. Takagi, H. Yasuura, and S. Yajima, "High-Speed VLSI Multiplication Algorithm with a Redundant Binary Addition Tree," *IEEE Trans. Computers,* vol. 34, no. 9, pp. 789-796, Sept. 1985.
[37] P.F. Williams, A. Biere, E.M. Clarke, and A. Gupta, "Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking," *Proc. 12th Int'l Conf. Computer Aided Verification (CAV '00),* pp. 124-138, 2000.
[38] H. Yu and J.A. Abraham, "An Efficient 3-Bit-Scan Multiplier without Overlapping Bits, and Its 64×64 Bit Implementation," *Proc. Seventh Asia and South Pacific Design Automation Conf. (ASP-DAC '02),* Jan. 2002.
[39] Z. Zhou, X. Song, F. Corella, E. Cerny, and M. Langevin, "Description and Verification of RTL Designs Using Multiway Decision Graphs," *Proc. IFIP Conf. Hardware Description Languages and Their Applications (CHDL '95),* 1995.
[40] Z. Zhou and W. Burleson, "Equivalence Checking of Datapaths Based on Canonical Arithmetic Expressions," *Proc. 32nd ACM/IEEE Conf. Design Automation (DAC '95),* pp. 546-551, 1995.

Shobha Vasudevan received the BE degree in computer engineering from the University of Mumbai in 2001 and the MSE degree in computer engineering from the University of Texas at Austin in 2003. She is a fourth year PhD student at the University of Texas at Austin. Her research interests are formal methods in system design and verification of hardware and embedded systems, model checking, term rewriting, and automata theory.

Vinod Viswanath received the MS and MPhil degrees from Yale University in 1999 and 2001, respectively. He is currently a PhD student in computer engineering at the University of Texas at Austin. He is also a formal verification research engineer at Intel Corp., Austin, Texas. His research interests lie broadly in the area of formal methods for hardware verification and low-power processor design.

Robert W. Sumners received the BSc, MSc, and PhD degrees in electrical engineering from the University of Texas at Austin with a technical focus in formal methods and the application and development of the ACL2 theorem prover. Since 1998, he has been an engineer working for Advanced Micro Devices in the areas of functional and formal verification of microprocessor designs. His research interests include the development of better algorithms and heuristics for improving the efficiency of theorem-proving tools and the use of theorem proving as a basis for sound reductions of the analysis of large-state systems to more tractable components. He is also researching the use of formal methods in developing state-based metrics for guiding the generation of functional verification patterns.

Jacob A. Abraham received the PhD degree in electrical engineering and computer science from Stanford University in 1974. He is a professor of electrical and computer engineering and a professor of computer sciences at the University of Texas at Austin. He is also the director of the Computer Engineering Research Center and holds a Cockrell Family Regents chair in engineering. He has published extensively and is included in a list of the most cited researchers in the world. He has supervised more than 60 PhD dissertations, and is particularly proud of the accomplishments of his students, many of whom occupy senior positions in academia and industry. His research interests include VLSI design and test, formal verification, and fault-tolerant computing. He has served as an associate editor of several IEEE transactions and as the chair of the IEEE CS Technical Committee on Fault-Tolerant Computing. He has been elected a fellow of the IEEE, as well as a fellow of the ACM. He is the recipient of the 2005 IEEE Emanuel R. Piore Award.