Automated and Scalable Verification of Integer Multipliers

Mertcan Temel[1,2]
mert@utexas.edu

Anna Slobodova[2]
anna@centtech.com

Warren A. Hunt, Jr.[1]
hunt@cs.utexas.edu

[1]University of Texas at Austin, Austin, TX, USA

[2]Centaur Technology, Inc., Austin, TX, USA

Jul 19-24, 2020 (CAV 2020)

Introduction

- Wallace-tree and Booth Encoding define algorithms to design efficient integer multipliers for hardware.
- Verification of these multipliers is a difficult problem:
  - SAT Solvers and BDDs do not scale.
  - Equivalence checking requires structurally close specifications.
  - Computer algebra methods perform better but with limitations.
- We propose a more efficient, rewrite-based method that is:
  - widely applicable (tested for 75+ benchmarks),
  - scalable (1024x1024-bit multipliers proved under 10 minutes),
  - provably correct (verified using ACL2)
- Our method works with RTL-level hierarchical designs

Introduction

- Wallace-tree and Booth Encoding define algorithms to design efficient integer multipliers for hardware.
- Verification of these multipliers is a difficult problem:
  - SAT Solvers and BDDs do not scale.
  - Equivalence checking requires structurally close specifications.
  - Computer algebra methods perform better but with limitations.
- We propose a more efficient, rewrite-based method that is:
  - widely applicable (tested for 75+ benchmarks),
  - scalable (1024x1024-bit multipliers proved under 10 minutes),
  - provably correct (verified using ACL2)
- Our method works with RTL-level hierarchical designs

Introduction

- Wallace-tree and Booth Encoding define algorithms to design efficient integer multipliers for hardware.
- Verification of these multipliers is a difficult problem:
  - SAT Solvers and BDDs do not scale.
  - Equivalence checking requires structurally close specifications.
  - Computer algebra methods perform better but with limitations.
- We propose a more efficient, rewrite-based method that is:
  - widely applicable (tested for 75+ benchmarks),
  - scalable (1024x1024-bit multipliers proved under 10 minutes),
  - provably correct (verified using ACL2)
- Our method works with RTL-level hierarchical designs

Introduction

- Wallace-tree and Booth Encoding define algorithms to design efficient integer multipliers for hardware.
- Verification of these multipliers is a difficult problem:
  - SAT Solvers and BDDs do not scale.
  - Equivalence checking requires structurally close specifications.
  - Computer algebra methods perform better but with limitations.
- We propose a more efficient, rewrite-based method that is:
  - widely applicable (tested for 75+ benchmarks),
  - scalable (1024x1024-bit multipliers proved under 10 minutes),
  - provably correct (verified using ACL2)
- Our method works with RTL-level hierarchical designs

Overview

1. Review of Integer Multipliers
2. The Method
3. Experiments

Review: Integer Multipliers

$$\begin{array}{cccccccc} & a_2 & a_2 & a_2 & a_2 & a_1 & a_0 \\ \mathbf{x} & b_2 & b_2 & b_2 & b_2 & b_1 & b_0 \\ \hline \end{array}$$

- Goal: implement an efficient hardware module that multiplies two bit-vectors.
- Integer multipliers have two stages:
  1. Partial Product Generation (e.g., Baugh-Wooley, Booth Encoding)
  2. Partial Product Summation (e.g., Array, Wallace-tree, Dadda-tree)
- Even two designs following the same algorithm may have very different structures. Therefore, it is important to have an automated system for verification.

Wallace-tree multiplication on 3x3-bit signed numbers

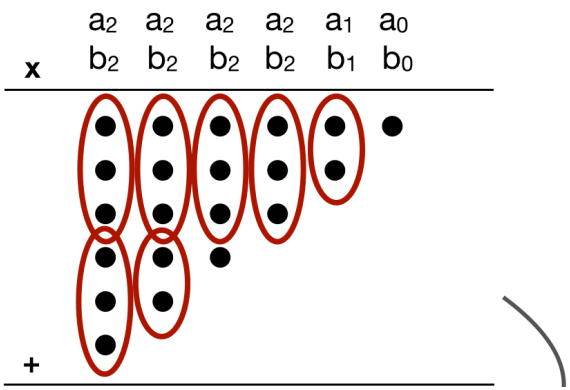
Review: Integer Multipliers

$$\begin{array}{cccccccc} & a_2 & a_2 & a_2 & a_2 & a_1 & a_0 \\ \mathbf{x} & b_2 & b_2 & b_2 & b_2 & b_1 & b_0 \\ \hline \end{array}$$

- Goal: implement an efficient hardware module that multiplies two bit-vectors.
- Integer multipliers have two stages:
  1. Partial Product Generation (e.g., Baugh-Wooley, Booth Encoding)
  2. Partial Product Summation (e.g., Array, Wallace-tree, Dadda-tree)
- Even two designs following the same algorithm may have very different structures. Therefore, it is important to have an automated system for verification.

Wallace-tree multiplication on 3x3-bit signed numbers

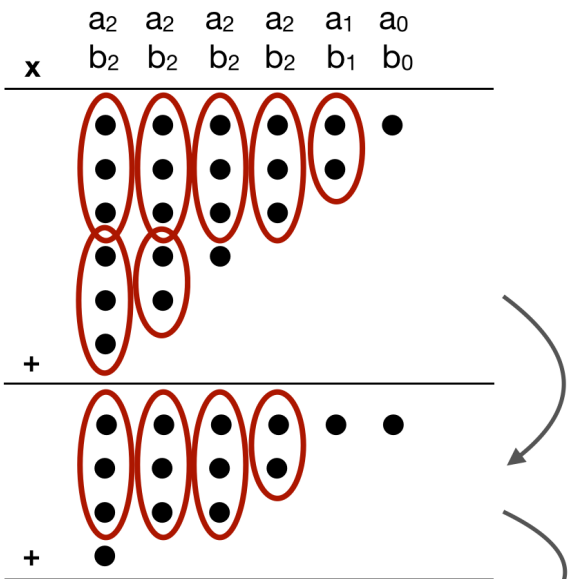
Review: Integer Multipliers



Wallace-tree multiplication
on 3x3-bit signed numbers

- Goal: implement an efficient hardware module that multiplies two bit-vectors.
- Integer multipliers have two stages:
  1. Partial Product Generation
     (e.g., Baugh-Wooley, Booth Encoding)
  2. Partial Product Summation
     (e.g., Array, Wallace-tree, Dadda-tree)
- Even two designs following the same algorithm may have very different structures. Therefore, it is important to have an automated system for verification.

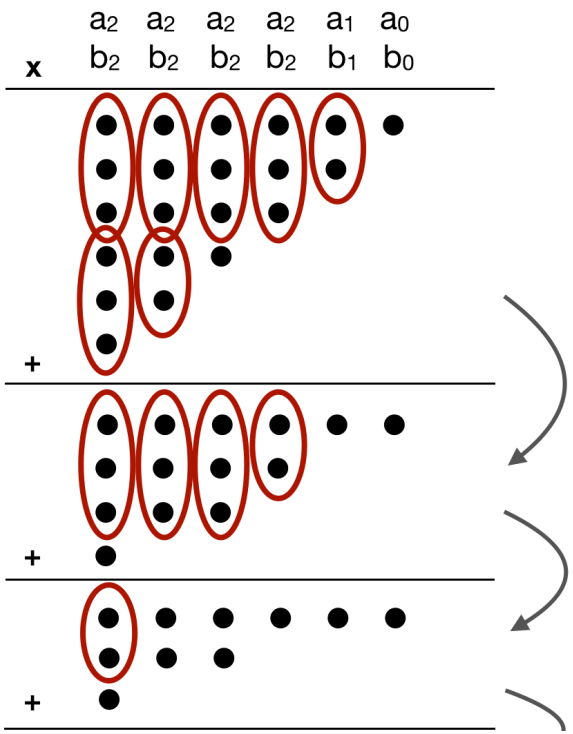
Review: Integer Multipliers



Wallace-tree multiplication on 3x3-bit signed numbers

- Goal: implement an efficient hardware module that multiplies two bit-vectors.
- Integer multipliers have two stages:
  1. Partial Product Generation (e.g., Baugh-Wooley, Booth Encoding)
  2. Partial Product Summation (e.g., Array, Wallace-tree, Dadda-tree)
- Even two designs following the same algorithm may have very different structures. Therefore, it is important to have an automated system for verification.

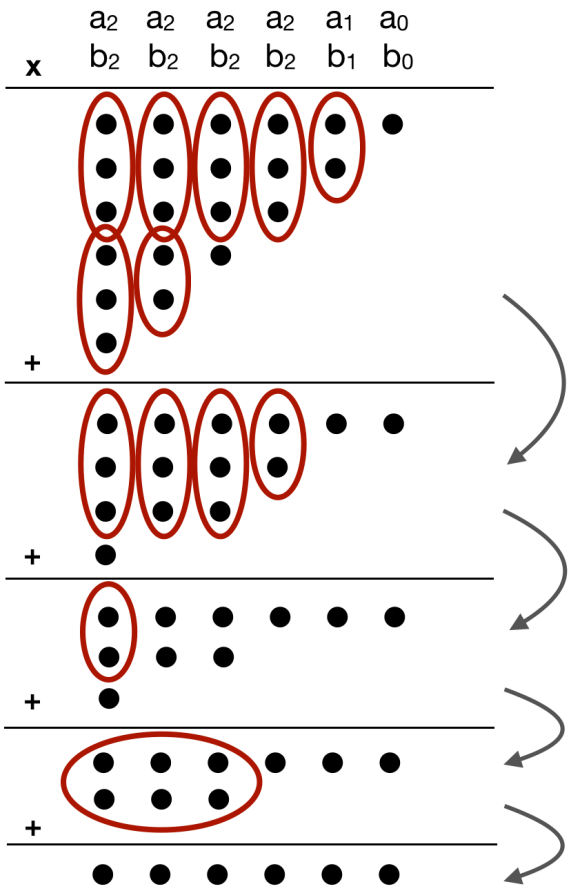
Review: Integer Multipliers



Wallace-tree multiplication
on 3x3-bit signed numbers

- Goal: implement an efficient hardware module that multiplies two bit-vectors.
- Integer multipliers have two stages:
  1. Partial Product Generation
     (e.g., Baugh-Wooley, Booth Encoding)
  2. Partial Product Summation
     (e.g., Array, Wallace-tree, Dadda-tree)
- Even two designs following the same algorithm may have very different structures. Therefore, it is important to have an automated system for verification.

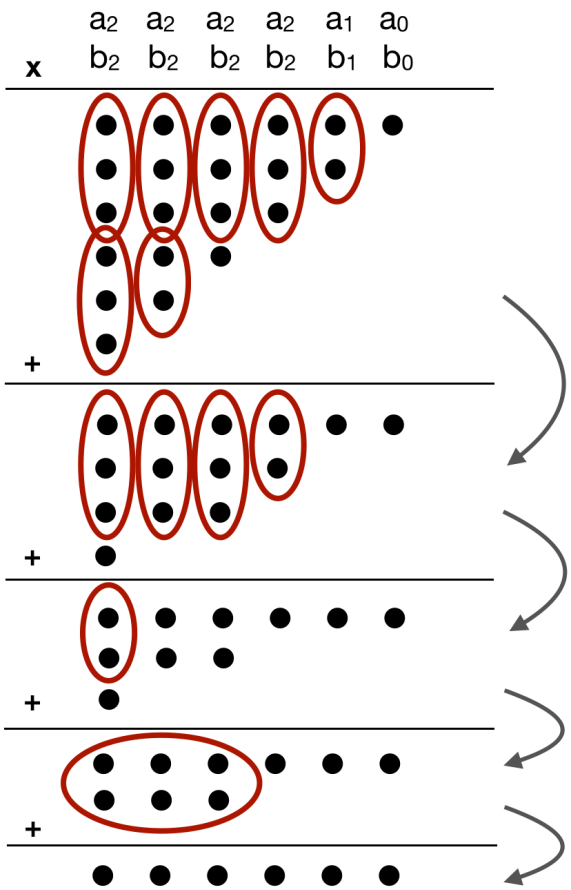
Review: Integer Multipliers



Wallace-tree multiplication on 3x3-bit signed numbers

- Goal: implement an efficient hardware module that multiplies two bit-vectors.
- Integer multipliers have two stages:
  1. Partial Product Generation (e.g., Baugh-Wooley, Booth Encoding)
  2. Partial Product Summation (e.g., Array, Wallace-tree, Dadda-tree)
- Even two designs following the same algorithm may have very different structures. Therefore, it is important to have an automated system for verification.

Review: Integer Multipliers



Wallace-tree multiplication on 3x3-bit signed numbers

- Goal: implement an efficient hardware module that multiplies two bit-vectors.
- Integer multipliers have two stages:
  1. Partial Product Generation (e.g., Baugh-Wooley, Booth Encoding)
  2. Partial Product Summation (e.g., Array, Wallace-tree, Dadda-tree)
- Even two designs following the same algorithm may have very different structures. Therefore, it is important to have an automated system for verification.

Specification

Our goal is to prove such conjectures:

```
(defthm multiplier_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <signed_mxn_mult>)
                  (truncate (+ M N)
                            (* (signext M a)
                               (signext N b))))))
```

- ACL2 is an interactive and automated theorem proving system.
- RTL Designs are translated from System Verilog to SVL format
- We prove that the circuit implements truncated multiplication of two sign-extended (or zero-extended) numbers.
- The mechanism we implemented to prove such conjectures are completely verified in ACL2.

Specification

Our goal is to prove such conjectures:

```
(defthm multiplier_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <signed_mxn_mult>)
                  (truncate (+ M N)
                            (* (signext M a)
                               (signext N b))))))
```

- ACL2 is an interactive and automated theorem proving system.
- RTL Designs are translated from System Verilog to SVL format
- We prove that the circuit implements truncated multiplication of two sign-extended (or zero-extended) numbers.
- The mechanism we implemented to prove such conjectures are completely verified in ACL2.

Specification

Our goal is to prove such conjectures:

```
(defthm multiplier_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <signed_mxn_mult>)
                  (truncate (+ M N)
                            (* (signext M a)
                               (signext N b))))))
```

- ACL2 is an interactive and automated theorem proving system.
- RTL Designs are translated from System Verilog to SVL format
- We prove that the circuit implements truncated multiplication of two sign-extended (or zero-extended) numbers.
- The mechanism we implemented to prove such conjectures are completely verified in ACL2.

Specification

Our goal is to prove such conjectures:

```
(defthm multiplier_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <signed_mxn_mult>)
                  (truncate (+ M N)
                            (* (signext M a)
                               (signext N b))))))
```

- ACL2 is an interactive and automated theorem proving system.
- RTL Designs are translated from System Verilog to SVL format
- We prove that the circuit implements truncated multiplication of two sign-extended (or zero-extended) numbers.
- The mechanism we implemented to prove such conjectures are completely verified in ACL2.

Specification

Our goal is to prove such conjectures:

```
(defthm multiplier_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <signed_mxn_mult>)
                  (truncate (+ M N)
                            (* (signext M a)
                               (signext N b))))))
```

- ACL2 is an interactive and automated theorem proving system.
- RTL Designs are translated from System Verilog to SVL format
- We prove that the circuit implements truncated multiplication of two sign-extended (or zero-extended) numbers.
- The mechanism we implemented to prove such conjectures are completely verified in ACL2.

Overview of the Method

Our goal is to prove such conjectures:

```
(defthm multiplier_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <signed_mxn_mult>)
                  (truncate (+ M N)
                            (* (signext M a)
                               (signext N b))))))
```

1. Before working on this conjecture, we reason about the adder modules
2. Then we submit the above event and:
   - Replace instantiations of adder modules with their specification from Step 1.
   - Simplify terms from summation tree algorithms.
   - Simplify terms from partial product generation algorithms.
   - Rewrite RHS to a form that syntactically matches the simplified form in LHS.

Overview of the Method

Our goal is to prove such conjectures:

```
(defthm multiplier_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <signed_mxn_mult>)
                  (truncate (+ M N)
                            (* (signext M a)
                               (signext N b))))))
```

1. Before working on this conjecture, we reason about the adder modules
2. Then we submit the above event and:
   - Replace instantiations of adder modules with their specification from Step 1.
   - Simplify terms from summation tree algorithms.
   - Simplify terms from partial product generation algorithms.
   - Rewrite RHS to a form that syntactically matches the simplified form in LHS.

Overview of the Method

Our goal is to prove such conjectures:

```
(defthm multiplier_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <signed_mxn_mult>)
                  (truncate (+ M N)
                            (* (signext M a)
                               (signext N b))))))
```

1. Before working on this conjecture, we reason about the adder modules
2. Then we submit the above event and:
   - Replace instantiations of adder modules with their specification from Step 1.
   - Simplify terms from summation tree algorithms.
   - Simplify terms from partial product generation algorithms.
   - Rewrite RHS to a form that syntactically matches the simplified form in LHS.

Overview of the Method

Our goal is to prove such conjectures:

```
(defthm multiplier_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <signed_mxn_mult>)
                  (truncate (+ M N)
                            (* (signext M a)
                               (signext N b))))))
```

1. Before working on this conjecture, we reason about the adder modules
2. Then we submit the above event and:
   - Replace instantiations of adder modules with their specification from Step 1.
   - Simplify terms from summation tree algorithms.
   - Simplify terms from partial product generation algorithms.
   - Rewrite RHS to a form that syntactically matches the simplified form in LHS.

Overview of the Method

Our goal is to prove such conjectures:

```
(defthm multiplier_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <signed_mxn_mult>)
                  (truncate (+ M N)
                            (* (signext M a)
                               (signext N b))))))
```

1. Before working on this conjecture, we reason about the adder modules
2. Then we submit the above event and:
   - Replace instantiations of adder modules with their specification from Step 1.
   - Simplify terms from summation tree algorithms.
   - Simplify terms from partial product generation algorithms.
   - Rewrite RHS to a form that syntactically matches the simplified form in LHS.

Overview of the Method

Our goal is to prove such conjectures:

```
(defthm multiplier_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <signed_mxn_mult>)
                  (truncate (+ M N)
                            (* (signext M a)
                               (signext N b))))))
```

1. Before working on this conjecture, we reason about the adder modules
2. Then we submit the above event and:
   - Replace instantiations of adder modules with their specification from Step 1.
   - Simplify terms from summation tree algorithms.
   - Simplify terms from partial product generation algorithms.
   - Rewrite RHS to a form that syntactically matches the simplified form in LHS.

Step 1: Adder Module Proofs

Before the multiplier proof, a lemma for each adder module is proved:

```
(defthm adder_module_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <an-adder-module>)
                  (spec-of-the-adder a b))))
```

Specification for some adders per output bit:

Adder	out_3	out_2	out_1	out_0
Half-adder	-	-	$c(a+b)$	$s(a+b)$
Full-adder	-	-	$c(a+b+c_{in})$	$s(a+b+c_{in})$
Vector adders	$s(a_3+b_3 \; +c(a_2+b_2 \; +c(a_1+b_1 \; +c(a_0+b_0)))$	$s(a_2+b_2 \; +c(a_1+b_1 \; +c(a_0+b_0)))$	$s(a_1+b_1 \; +c(a_0+b_0))$	$s(a_0+b_0)$

where $s(x) = mod_2(x)$ and $c(x) = \left\lfloor \frac{x}{2} \right\rfloor$

Step 1: Adder Module Proofs

Before the multiplier proof, a lemma for each adder module is proved:

```
(defthm adder_module_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <an-adder-module>)
                  (spec-of-the-adder a b))))
```

Specification for some adders per output bit:

Adder	out_3	out_2	out_1	out_0
Half-adder	-	-	$c(a+b)$	$s(a+b)$
Full-adder	-	-	$c(a+b+c_{in})$	$s(a+b+c_{in})$
Vector adders	$s(a_3+b_3 \; +c(a_2+b_2 \; +c(a_1+b_1 \; +c(a_0+b_0)))$	$s(a_2+b_2 \; +c(a_1+b_1 \; +c(a_0+b_0)))$	$s(a_1+b_1 \; +c(a_0+b_0))$	$s(a_0+b_0)$

where $s(x) = mod_2(x)$ and $c(x) = \left\lfloor \frac{x}{2} \right\rfloor$

Step 1: Adder Module Proofs

Before the multiplier proof, a lemma for each adder module is proved:

```
(defthm adder_module_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <an-adder-module>)
                  (spec-of-the-adder a b))))
```

Specification for some adders per output bit:

Adder	out_3	out_2	out_1	out_0
Half-adder	-	-	$c(a+b)$	$s(a+b)$
Full-adder	-	-	$c(a+b+c_{in})$	$s(a+b+c_{in})$
Vector adders	$s(a_3+b_3$ $+c(a_2+b_2$ $+c(a_1+b_1$ $+c(a_0+b_0)))$	$s(a_2+b_2$ $+c(a_1+b_1$ $+c(a_0+b_0)))$	$s(a_1+b_1$ $+c(a_0+b_0))$	$s(a_0+b_0)$

where $s(x) = mod_2(x)$ and $c(x) = \left\lfloor \frac{x}{2} \right\rfloor$

Step 1: Adder Module Proofs

Before the multiplier proof, a lemma for each adder module is proved:

```
(defthm adder_module_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <an-adder-module>)
                  (spec-of-the-adder a b))))
```

Specification for some adders per output bit:

Adder	out_3	out_2	out_1	out_0
Half-adder	-	-	$\mathbf{c(a+b)}$	$\mathbf{s(a+b)}$
Full-adder	-	-	$c(a+b+c_{in})$	$s(a+b+c_{in})$
Vector adders	$s(a_3+b_3$ $+c(a_2+b_2$ $+c(a_1+b_1$ $+c(a_0+b_0)))$	$s(a_2+b_2$ $+c(a_1+b_1$ $+c(a_0+b_0)))$	$s(a_1+b_1$ $+c(a_0+b_0))$	$s(a_0+b_0)$

where $s(x) = mod_2(x)$ and $c(x) = \left\lfloor \frac{x}{2} \right\rfloor$

Step 1: Adder Module Proofs

Before the multiplier proof, a lemma for each adder module is proved:

```
(defthm adder_module_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <an-adder-module>)
                  (spec-of-the-adder a b))))
```

Specification for some adders per output bit:

Adder	out_3	out_2	out_1	out_0
Half-adder	-	-	$c(a+b)$	$s(a+b)$
Full-adder	-	-	$\mathbf{c(a+b+c_{in})}$	$\mathbf{s(a+b+c_{in})}$
Vector adders	$s(a_3+b_3 + c(a_2+b_2 + c(a_1+b_1 + c(a_0+b_0)))$	$s(a_2+b_2 + c(a_1+b_1 + c(a_0+b_0)))$	$s(a_1+b_1 + c(a_0+b_0))$	$s(a_0+b_0)$

where $s(x) = mod_2(x)$ and $c(x) = \left\lfloor \frac{x}{2} \right\rfloor$

Step 1: Adder Module Proofs

Before the multiplier proof, a lemma for each adder module is proved:

```
(defthm adder_module_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <an-adder-module>)
                  (spec-of-the-adder a b))))
```

Specification for some adders per output bit:

Adder	out_3	out_2	out_1	out_0
Half-adder	-	-	$c(a+b)$	$s(a+b)$
Full-adder	-	-	$c(a+b+c_{in})$	$s(a+b+c_{in})$
Vector adders	$\mathbf{s(a_3+b_3}$ $\mathbf{+c(a_2+b_2}$ $\mathbf{+c(a_1+b_1}$ $\mathbf{+c(a_0+b_0))}$	$\mathbf{s(a_2+b_2}$ $\mathbf{+c(a_1+b_1}$ $\mathbf{+c(a_0+b_0)))}$	$\mathbf{s(a_1+b_1}$ $\mathbf{+c(a_0+b_0))}$	$\mathbf{s(a_0+b_0)}$

where $s(x) = mod_2(x)$ and $c(x) = \left\lfloor \frac{x}{2} \right\rfloor$

Step 1: Adder Module Proofs

Before the multiplier proof, a lemma for each adder module is proved:

```
(defthm adder_module_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <an-adder-module>)
                  (spec-of-the-adder a b))))
```

Specification for some adders per output bit:

Adder	out_3	out_2	out_1	out_0
Half-adder	-	-	$c(a+b)$	$s(a+b)$
Full-adder	-	-	$c(a+b+c_{in})$	$s(a+b+c_{in})$
Vector adders	$s(a_3+b_3$ $+c(a_2+b_2$ $+c(a_1+b_1$ $+c(a_0+b_0)))$	$s(a_2+b_2$ $+c(a_1+b_1$ $+c(a_0+b_0)))$	$s(a_1+b_1$ $+c(a_0+b_0))$	$s(a_0+b_0)$

where $s(x) = mod_2(x)$ and $c(x) = \left\lfloor \frac{x}{2} \right\rfloor$

Step 2: Multiplier Module Proofs

Prove this conjecture:

```
(defthm multiplier_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <signed_mxn_mult>)
                  (truncate (+ M N)
                            (* (signext M a)
                               (signext N b))))))
```

Both LHS and RHS should be rewritten to the same final form.

An example final form for the first 4 output bits of a multiplier:

out_3	out_2	out_1	out_0
$s(a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0 + c(a_0b_2 + a_1b_1 + a_2b_0 + c(a_1b_0 + a_0b_1 + c(a_0b_0)))$	$s(a_0b_2 + a_1b_1 + a_2b_0 + c(a_1b_0 + a_0b_1 + c(a_0b_0)))$	$s(a_1b_0 + a_0b_1 + c(a_0b_0))$	$s(a_0b_0)$

Step 2: Multiplier Module Proofs

Prove this conjecture:

```
(defthm multiplier_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <signed_mxn_mult>)
                  (truncate (+ M N)
                            (* (signext M a)
                               (signext N b))))))
```

Both LHS and RHS should be rewritten to the same final form.

An example final form for the first 4 output bits of a multiplier:

out_3	out_2	out_1	out_0
$s(a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0 + c(a_0b_2 + a_1b_1 + a_2b_0 + c(a_1b_0 + a_0b_1 + c(a_0b_0))))$	$s(a_0b_2 + a_1b_1 + a_2b_0 + c(a_1b_0 + a_0b_1 + c(a_0b_0)))$	$s(a_1b_0 + a_0b_1 + c(a_0b_0))$	$s(a_0b_0)$

Step 2: Multiplier Module Proofs

Prove this conjecture:

```
(defthm multiplier_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <signed_mxn_mult>)
                  (truncate (+ M N)
                            (* (signext M a)
                               (signext N b))))))
```

Both LHS and RHS should be rewritten to the same final form.

An example final form for the first 4 output bits of a multiplier:

out_3	out_2	out_1	out_0
$s(a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0$ $+c(a_0b_2 + a_1b_1 + a_2b_0$ $+c(a_1b_0 + a_0b_1$ $+c(a_0b_0)))$	$s(a_0b_2 + a_1b_1 + a_2b_0$ $+c(a_1b_0 + a_0b_1$ $+c(a_0b_0)))$	$s(a_1b_0 + a_0b_1$ $+c(a_0b_0))$	$s(a_0b_0)$

Step 2: Multiplier Module Proofs

Prove this conjecture:

```
(defthm multiplier_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <signed_mxn_mult>)
                  (truncate (+ M N)
                            (* (signext M a)
                               (signext N b))))))
```

Both LHS and RHS should be rewritten to the same final form.

An example final form for the first 4 output bits of a multiplier:

out_3	out_2	out_1	out_0
$s(a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0$ $+c(a_0b_2 + a_1b_1 + a_2b_0$ $+c(a_1b_0 + a_0b_1$ $+c(a_0b_0)))$	$s(a_0b_2 + a_1b_1 + a_2b_0$ $+c(a_1b_0 + a_0b_1$ $+c(a_0b_0)))$	$s(a_1b_0 + a_0b_1$ $+c(a_0b_0))$	$s(a_0b_0)$

Step 2: Multiplier Module Proofs

Prove this conjecture:

```
(defthm multiplier_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <signed_mxn_mult>)
                  (truncate (+ M N)
                            (* (signext M a)
                               (signext N b))))))
```

Both LHS and RHS should be rewritten to the same final form.

An example final form for the first 4 output bits of a multiplier:

out_3	out_2	out_1	out_0
$s(a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0$ $+c(a_0b_2 + a_1b_1 + a_2b_0$ $+c(a_1b_0 + a_0b_1$ $+c(a_0b_0)))$	$s(a_0b_2 + a_1b_1 + a_2b_0$ $+c(a_1b_0 + a_0b_1$ $+c(a_0b_0)))$	$s(a_1b_0 + a_0b_1$ $+c(a_0b_0))$	$s(a_0b_0)$

Step 2: Multiplier Module Proofs

Prove this conjecture:

```
(defthm multiplier_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <signed_mxn_mult>)
                  (truncate (+ M N)
                            (* (signext M a)
                               (signext N b))))))
```

Both LHS and RHS should be rewritten to the same final form.

An example final form for the first 4 output bits of a multiplier:

out_3	out_2	out_1	out_0
$s(a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0$ $+c(a_0b_2 + a_1b_1 + a_2b_0$ $+c(a_1b_0 + a_0b_1$ $+c(a_0b_0)))$	$s(a_0b_2 + a_1b_1 + a_2b_0$ $+c(a_1b_0 + a_0b_1$ $+c(a_0b_0)))$	$s(a_1b_0 + a_0b_1$ $+c(a_0b_0))$	$s(a_0b_0)$

Step 2: Multiplier Module Proofs

Prove this conjecture:

```
(defthm multiplier_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <signed_mxn_mult>)
                  (truncate (+ M N)
                            (* (signext M a)
                               (signext N b))))))
```

Both LHS and RHS should be rewritten to the same final form.

An example final form for the first 4 output bits of a multiplier:

out_3	out_2	out_1	out_0
$s(a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0$ $+c(a_0b_2 + a_1b_1 + a_2b_0$ $+c(a_1b_0 + a_0b_1$ $+c(a_0b_0)))$	$s(a_0b_2 + a_1b_1 + a_2b_0$ $+c(a_1b_0 + a_0b_1$ $+c(a_0b_0)))$	$s(a_1b_0 + a_0b_1$ $+c(a_0b_0))$	$s(a_0b_0)$

Step 2a: Simplify Summation Trees

The 4th LSB of the Wallace-tree multiplier
with simple partial products:

$$\begin{aligned} & s(\ s(\ s(a_3 b_0 + a_2 b_1 + a_1 b_2) \\ & \qquad + a_0 b_3 \\ & \qquad + c(a_2 b_0 + a_1 b_1 + a_0 b_2)) \\ & \quad + c(s(a_2 b_0 + a_1 b_1 + a_0 b_2) + c(a_1 b_0 + a_0 b_1))) \end{aligned}$$

Goal: Simplify such terms with a set of lemmas.

Nested instances of s can be cleared with the following lemma.

Lemma $\forall x, y \in \mathbb{Z}\ s(s(x) + y) = s(x + y)$

When this lemma applied to the example above, we get:

$$\begin{aligned} & s(a_3 b_0 + a_2 b_1 + a_1 b_2 + a_0 b_3 \\ & \quad + c(a_2 b_0 + a_1 b_1 + a_0 b_2) \\ & \quad + c(s(a_2 b_0 + a_1 b_1 + a_0 b_2) + c(a_1 b_0 + a_0 b_1))) \end{aligned}$$

Step 2a: Simplify Summation Trees

The 4th LSB of the Wallace-tree multiplier
with simple partial products:

$$\begin{aligned} s(\ & s(\ s(a_3b_0 + a_2b_1 + a_1b_2) \\ & \quad + a_0b_3 \\ & \quad + c(a_2b_0 + a_1b_1 + a_0b_2)) \\ & + c(s(a_2b_0 + a_1b_1 + a_0b_2) + c(a_1b_0 + a_0b_1))) \end{aligned}$$

Goal: Simplify such terms with a set of lemmas.

Nested instances of s can be cleared with the following lemma.

Lemma $\forall x, y \in \mathbb{Z}\ s(s(x) + y) = s(x + y)$

When this lemma applied to the example above, we get:

$$\begin{aligned} & s(a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 \\ & \quad + c(a_2b_0 + a_1b_1 + a_0b_2) \\ & \quad + c(s(a_2b_0 + a_1b_1 + a_0b_2) + c(a_1b_0 + a_0b_1))) \end{aligned}$$

Step 2a: Simplify Summation Trees

The 4th LSB of the Wallace-tree multiplier
with simple partial products:

$$
\begin{aligned}
&s(\ s(\ s(\ a_3b_0 + a_2b_1 + a_1b_2) \\
&\quad + a_0b_3 \\
&\quad + c(a_2b_0 + a_1b_1 + a_0b_2)) \\
&\ + c(s(a_2b_0 + a_1b_1 + a_0b_2) + c(a_1b_0 + a_0b_1)))
\end{aligned}
$$

Goal: Simplify such terms with a set of lemmas.

Nested instances of s can be cleared with the following lemma.

Lemma $\forall x, y \in \mathbb{Z}\ s(s(x) + y) = s(x + y)$

When this lemma applied to the example above, we get:

$$
\begin{aligned}
&s(a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 \\
&\ + c(a_2b_0 + a_1b_1 + a_0b_2) \\
&\ + c(s(a_2b_0 + a_1b_1 + a_0b_2) + c(a_1b_0 + a_0b_1)))
\end{aligned}
$$

Step 2a: Simplify Summation Trees

The 4th LSB of the Wallace-tree multiplier
with simple partial products:

$$
\begin{aligned}
&s(\ s(\ s(\ a_3 b_0 + a_2 b_1 + a_1 b_2) \\
&\quad + a_0 b_3 \\
&\quad + c(a_2 b_0 + a_1 b_1 + a_0 b_2)) \\
&\ + c(s(a_2 b_0 + a_1 b_1 + a_0 b_2) + c(a_1 b_0 + a_0 b_1)))
\end{aligned}
$$

Goal: Simplify such terms with a set of lemmas.

Nested instances of s can be cleared with the following lemma.

Lemma $\forall x, y \in \mathbb{Z}\ s(s(x) + y) = s(x + y)$

When this lemma applied to the example above, we get:

$$
\begin{aligned}
&s(a_3 b_0 + a_2 b_1 + a_1 b_2 + a_0 b_3 \\
&\ + c(a_2 b_0 + a_1 b_1 + a_0 b_2) \\
&\ + c(s(a_2 b_0 + a_1 b_1 + a_0 b_2) + c(a_1 b_0 + a_0 b_1)))
\end{aligned}
$$

Step 2a: Simplify Summation Trees (cntd.)

The current term:

$$s(a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 \\ +c(a_2b_0 + a_1b_1 + a_0b_2) \\ +c(s(a_2b_0 + a_1b_1 + a_0b_2) + c(a_1b_0 + a_0b_1)))$$

Define $d(x) = \frac{x}{2}$. Summation of two or more c instances can be merged with the following set of lemmas.

Lemma $\forall x, y \in \mathbb{Z}\ c(x) + c(y) = d(x + y - s(x) - s(y))$
Lemma $\forall x, y \in \mathbb{Z}\ c(x) + d(y) = d(x + y - s(x))$
Lemma $\forall x, y \in \mathbb{Z}\ d(x) + d(y) = d(x + y)$
Lemma $\forall x \in \mathbb{Z}\ d(-s(x) + x) = c(x)$

When these lemmas are applied to the example above, we get:

$$s(a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 \\ +c(a_2b_0 + a_1b_1 + a_0b_2 \\ +c(a_1b_0 + a_0b_1)))$$

This matches the final form. This works for much larger terms as well.

Step 2a: Simplify Summation Trees (cntd.)

The current term:

$$s(a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 \\ + c\,(a_2b_0 + a_1b_1 + a_0b_2) \\ + c\,(s(a_2b_0 + a_1b_1 + a_0b_2) + c(a_1b_0 + a_0b_1)))$$

Define $d(x) = \frac{x}{2}$. Summation of two or more c instances can be merged with the following set of lemmas.

Lemma $\forall x, y \in \mathbb{Z}\ c(x) + c(y) = d(x + y - s(x) - s(y))$
Lemma $\forall x, y \in \mathbb{Z}\ c(x) + d(y) = d(x + y - s(x))$
Lemma $\forall x, y \in \mathbb{Z}\ d(x) + d(y) = d(x + y)$
Lemma $\forall x \in \mathbb{Z}\ d(-s(x) + x) = c(x)$

When these lemmas are applied to the example above, we get:

$$s(a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 \\ + c(a_2b_0 + a_1b_1 + a_0b_2 \\ + c(a_1b_0 + a_0b_1)))$$

This matches the final form. This works for much larger terms as well.

Step 2a: Simplify Summation Trees (cntd.)

The current term:

$$s(a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 \\ + c\,(a_2b_0 + a_1b_1 + a_0b_2) \\ + c\,(s(a_2b_0 + a_1b_1 + a_0b_2) + c(a_1b_0 + a_0b_1)))$$

Define $d(x) = \frac{x}{2}$. Summation of two or more c instances can be merged with the following set of lemmas.

Lemma $\forall x, y \in \mathbb{Z}\ c(x) + c(y) = d(x + y - s(x) - s(y))$
Lemma $\forall x, y \in \mathbb{Z}\ c(x) + d(y) = d(x + y - s(x))$
Lemma $\forall x, y \in \mathbb{Z}\ d(x) + d(y) = d(x + y)$
Lemma $\forall x \in \mathbb{Z}\ d(-s(x) + x) = c(x)$

When these lemmas are applied to the example above, we get:

$$s(a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 \\ + c(a_2b_0 + a_1b_1 + a_0b_2 \\ + c(a_1b_0 + a_0b_1)))$$

This matches the final form. This works for much larger terms as well.

Step 2a: Simplify Summation Trees (cntd.)

The current term:

$$s(a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 \\ + c\,(a_2b_0 + a_1b_1 + a_0b_2) \\ + c\,(s(a_2b_0 + a_1b_1 + a_0b_2) + c(a_1b_0 + a_0b_1)))$$

Define $d(x) = \frac{x}{2}$. Summation of two or more c instances can be merged with the following set of lemmas.

Lemma $\forall x, y \in \mathbb{Z}\ c(x) + c(y) = d(x + y - s(x) - s(y))$
Lemma $\forall x, y \in \mathbb{Z}\ c(x) + d(y) = d(x + y - s(x))$
Lemma $\forall x, y \in \mathbb{Z}\ d(x) + d(y) = d(x + y)$
Lemma $\forall x \in \mathbb{Z}\ d(-s(x) + x) = c(x)$

When these lemmas are applied to the example above, we get:

$$s(a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 \\ + c(a_2b_0 + a_1b_1 + a_0b_2 \\ + c(a_1b_0 + a_0b_1)))$$

This matches the final form. This works for much larger terms as well.

Step 2a: Simplify Summation Trees (cntd.)

The current term:

$$s(a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 \\ + c\,(a_2b_0 + a_1b_1 + a_0b_2) \\ + c\,(s(a_2b_0 + a_1b_1 + a_0b_2) + c(a_1b_0 + a_0b_1)))$$

Define $d(x) = \frac{x}{2}$. Summation of two or more c instances can be merged with the following set of lemmas.

Lemma $\forall x, y \in \mathbb{Z}\ c(x) + c(y) = d(x + y - s(x) - s(y))$
Lemma $\forall x, y \in \mathbb{Z}\ c(x) + d(y) = d(x + y - s(x))$
Lemma $\forall x, y \in \mathbb{Z}\ d(x) + d(y) = d(x + y)$
Lemma $\forall x \in \mathbb{Z}\ d(-s(x) + x) = c(x)$

When these lemmas are applied to the example above, we get:

$$s(a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 \\ + c(a_2b_0 + a_1b_1 + a_0b_2 \\ + c(a_1b_0 + a_0b_1)))$$

This matches the final form. This works for much larger terms as well.

Step 2b: Simplify Partial Products

Booth Encoding creates more complicated terms for partial products.

For example:

$$s([\neg b_1 b_0 a_1 \vee b_1 \neg b_0 \neg a_0 \vee b_1 b_0 \neg a_1] + c([b_1 b_0 \vee b_1 \neg b_0] + [b_1 \neg b_0 \vee \neg b_1 b_0 a_0 \vee b_1 b_0 \neg a_0]))$$

First, we perform algebraic rewriting to get rid of $\oplus$, $\vee$ and $\neg$.

Lemma $\forall x \in \{0,1\}\ \neg x = 1 - x$
Lemma $\forall x, y \in \{0,1\}\ x \vee y = x + y - xy$
Lemma $\forall x, y \in \{0,1\}\ x \oplus y = x + y - xy\ \ - xy$

When these lemmas are applied to the example above, we get:

$$s(b_1 + b_0 a_1 - b_1 a_0 + b_1 b_0 a_0 - b_1 b_0 a_1 - b_1 b_0 a_1 + c(b_1 + b_1 + b_0 a_0 - b_1 b_0 a_0 - b_1 b_0 a_0))$$

Step 2b: Simplify Partial Products

Booth Encoding creates more complicated terms for partial products.

For example:

$$s([\neg b_1 b_0 a_1 \vee b_1 \neg b_0 \neg a_0 \vee b_1 b_0 \neg a_1] + c([b_1 b_0 \vee b_1 \neg b_0] + [b_1 \neg b_0 \vee \neg b_1 b_0 a_0 \vee b_1 b_0 \neg a_0]))$$

First, we perform algebraic rewriting to get rid of $\oplus$, $\vee$ and $\neg$.

Lemma $\forall x \in \{0,1\}\ \neg x = 1 - x$
Lemma $\forall x, y \in \{0,1\}\ x \vee y = x + y - xy$
Lemma $\forall x, y \in \{0,1\}\ x \oplus y = x + y - xy\ \ - xy$

When these lemmas are applied to the example above, we get:

$$s(b_1 + b_0 a_1 - b_1 a_0 + b_1 b_0 a_0 - b_1 b_0 a_1 - b_1 b_0 a_1 + c(b_1 + b_1 + b_0 a_0 - b_1 b_0 a_0 - b_1 b_0 a_0))$$

Step 2b: Simplify Partial Products

Booth Encoding creates more complicated terms for partial products.

For example:

$$s([\neg b_1 b_0 a_1 \vee b_1 \neg b_0 \neg a_0 \vee b_1 b_0 \neg a_1] + c([b_1 b_0 \vee b_1 \neg b_0] + [b_1 \neg b_0 \vee \neg b_1 b_0 a_0 \vee b_1 b_0 \neg a_0]))$$

First, we perform algebraic rewriting to get rid of $\oplus$, $\vee$ and $\neg$.

Lemma $\forall x \in \{0,1\}\ \neg x = 1 - x$
Lemma $\forall x, y \in \{0,1\}\ x \vee y = x + y - xy$
Lemma $\forall x, y \in \{0,1\}\ x \oplus y = x + y - xy\ \ - xy$

When these lemmas are applied to the example above, we get:

$$s(b_1 + b_0 a_1 - b_1 a_0 + b_1 b_0 a_0 - b_1 b_0 a_1 - b_1 b_0 a_1 + c(b_1 + b_1 + b_0 a_0 - b_1 b_0 a_0 - b_1 b_0 a_0))$$

Step 2b: Simplify Partial Products

Booth Encoding creates more complicated terms for partial products.

For example:

$$s([\neg b_1 b_0 a_1 \vee b_1 \neg b_0 \neg a_0 \vee b_1 b_0 \neg a_1] + c([b_1 b_0 \vee b_1 \neg b_0] + [b_1 \neg b_0 \vee \neg b_1 b_0 a_0 \vee b_1 b_0 \neg a_0]))$$

First, we perform algebraic rewriting to get rid of $\oplus$, $\vee$ and $\neg$.

Lemma $\forall x \in \{0,1\}\ \neg x = 1 - x$
Lemma $\forall x, y \in \{0,1\}\ x \vee y = x + y - xy$
Lemma $\forall x, y \in \{0,1\}\ x \oplus y = x + y - xy\ \ - xy$

When these lemmas are applied to the example above, we get:

$$s(b_1 + b_0 a_1 - b_1 a_0 + b_1 b_0 a_0 - b_1 b_0 a_1 - b_1 b_0 a_1 + c(b_1 + b_1 + b_0 a_0 - b_1 b_0 a_0 - b_1 b_0 a_0))$$

Step 2b: Simplify Partial Products (cntd.)

The current term:

$$s(b_1 + b_0 a_1 - b_1 a_0 + b_1 b_0 a_0 - b_1 b_0 a_1 - b_1 b_0 a_1 \\ + c(b_1 + b_1 + b_0 a_0 - b_1 b_0 a_0 - b_1 b_0 a_0))$$

We try to get rid of repeated and/or negative minterms with the following lemmas.

Lemma $\forall x, y \in \mathbb{Z}\ s((-x) + y) = s(x + y)$
Lemma $\forall x, y \in \mathbb{Z}\ c((-x) + y) = (-x) + c(x + y)$
Lemma $\forall x, y \in \mathbb{Z}\ d((-x) + y) = (-x) + d(x + y)$

Lemma $\forall x, y \in \mathbb{Z}\ s(x + x + y) = s(y)$
Lemma $\forall x, y \in \mathbb{Z}\ c(x + x + y) = x + c(y)$
Lemma $\forall x, y \in \mathbb{Z}\ d(x + x + y) = x + d(y)$

When these lemmas are applied to the example above, we get:

$$s(b_0 a_1 + b_1 a_0 + c(b_0 a_0))$$

Let's see the experiment results!

Step 2b: Simplify Partial Products (cntd.)

The current term:

$$s(b_1 + b_0 a_1 - b_1 a_0 + b_1 b_0 a_0 - b_1 b_0 a_1 - b_1 b_0 a_1 \\ + c(b_1 + b_1 + b_0 a_0 - b_1 b_0 a_0 - b_1 b_0 a_0))$$

We try to get rid of repeated and/or negative minterms with the following lemmas.

Lemma $\forall x, y \in \mathbb{Z}\ s((-x) + y) = s(x + y)$
Lemma $\forall x, y \in \mathbb{Z}\ c((-x) + y) = (-x) + c(x + y)$
Lemma $\forall x, y \in \mathbb{Z}\ d((-x) + y) = (-x) + d(x + y)$
Lemma $\forall x, y \in \mathbb{Z}\ s(x + x + y) = s(y)$
Lemma $\forall x, y \in \mathbb{Z}\ c(x + x + y) = x + c(y)$
Lemma $\forall x, y \in \mathbb{Z}\ d(x + x + y) = x + d(y)$

When these lemmas are applied to the example above, we get:

$$s(b_0 a_1 + b_1 a_0 + c(b_0 a_0))$$

Let's see the experiment results!

Step 2b: Simplify Partial Products (cntd.)

The current term:

$$s(b_1 + b_0 a_1 - b_1 a_0 + b_1 b_0 a_0 - b_1 b_0 a_1 - b_1 b_0 a_1 \\ + c(b_1 + b_1 + b_0 a_0 - b_1 b_0 a_0 - b_1 b_0 a_0))$$

We try to get rid of repeated and/or negative minterms with the following lemmas.

Lemma $\forall x, y \in \mathbb{Z}\ s((-x) + y) = s(x + y)$
Lemma $\forall x, y \in \mathbb{Z}\ c((-x) + y) = (-x) + c(x + y)$
Lemma $\forall x, y \in \mathbb{Z}\ d((-x) + y) = (-x) + d(x + y)$

Lemma $\forall x, y \in \mathbb{Z}\ s(x + x + y) = s(y)$
Lemma $\forall x, y \in \mathbb{Z}\ c(x + x + y) = x + c(y)$
Lemma $\forall x, y \in \mathbb{Z}\ d(x + x + y) = x + d(y)$

When these lemmas are applied to the example above, we get:

$$s(b_0 a_1 + b_1 a_0 + c(b_0 a_0))$$

Let's see the experiment results!

Step 2b: Simplify Partial Products (cntd.)

The current term:

$$s(b_1 + b_0 a_1 - b_1 a_0 + b_1 b_0 a_0 - b_1 b_0 a_1 - b_1 b_0 a_1 \\ + c(b_1 + b_1 + b_0 a_0 - b_1 b_0 a_0 - b_1 b_0 a_0))$$

We try to get rid of repeated and/or negative minterms with the following lemmas.

Lemma $\forall x, y \in \mathbb{Z}\ s((-x) + y) = s(x + y)$
Lemma $\forall x, y \in \mathbb{Z}\ c((-x) + y) = (-x) + c(x + y)$
Lemma $\forall x, y \in \mathbb{Z}\ d((-x) + y) = (-x) + d(x + y)$

Lemma $\forall x, y \in \mathbb{Z}\ s(x + x + y) = s(y)$
Lemma $\forall x, y \in \mathbb{Z}\ c(x + x + y) = x + c(y)$
Lemma $\forall x, y \in \mathbb{Z}\ d(x + x + y) = x + d(y)$

When these lemmas are applied to the example above, we get:

$$s(b_0 a_1 + b_1 a_0 + c(b_0 a_0))$$

Let's see the experiment results!

Step 2b: Simplify Partial Products (cntd.)

The current term:

$$s(b_1 + b_0 a_1 - b_1 a_0 + b_1 b_0 a_0 - b_1 b_0 a_1 - b_1 b_0 a_1 \\ + c(b_1 + b_1 + b_0 a_0 - b_1 b_0 a_0 - b_1 b_0 a_0))$$

We try to get rid of repeated and/or negative minterms with the following lemmas.

Lemma $\forall x, y \in \mathbb{Z}\ s((-x) + y) = s(x + y)$
Lemma $\forall x, y \in \mathbb{Z}\ c((-x) + y) = (-x) + c(x + y)$
Lemma $\forall x, y \in \mathbb{Z}\ d((-x) + y) = (-x) + d(x + y)$

Lemma $\forall x, y \in \mathbb{Z}\ s(x + x + y) = s(y)$
Lemma $\forall x, y \in \mathbb{Z}\ c(x + x + y) = x + c(y)$
Lemma $\forall x, y \in \mathbb{Z}\ d(x + x + y) = x + d(y)$

When these lemmas are applied to the example above, we get:

$$s(b_0 a_1 + b_1 a_0 + c(b_0 a_0))$$

Let's see the experiment results!

Experiments

Our method is tested for various integer multipliers with:

- Signed/unsigned simple or Booth Encoded partial products.
- Summation trees such as Wallace-tree, Dadda-tree, 4:2 compressor trees...
- Final stage adders such as Brent-Kung, Ladner-Fischer, Carry-lookahead...

We measure the total time for:

- Proof events for each adder module in Step 1
- Proof event for the multiplier module in Step 2

Experiments

Our method is tested for various integer multipliers with:

- Signed/unsigned simple or Booth Encoded partial products.
- Summation trees such as Wallace-tree, Dadda-tree, 4:2 compressor trees...
- Final stage adders such as Brent-Kung, Ladner-Fischer, Carry-lookahead...

We measure the total time for:

- Proof events for each adder module in Step 1
- Proof event for the multiplier module in Step 2

Experiments

Our method is tested for various integer multipliers with:

- Signed/unsigned simple or Booth Encoded partial products.
- Summation trees such as Wallace-tree, Dadda-tree, 4:2 compressor trees...
- Final stage adders such as Brent-Kung, Ladner-Fischer, Carry-lookahead...

We measure the total time for:

- Proof events for each adder module in Step 1
- Proof event for the multiplier module in Step 2

Experiments

Our method is tested for various integer multipliers with:

- Signed/unsigned simple or Booth Encoded partial products.
- Summation trees such as Wallace-tree, Dadda-tree, 4:2 compressor trees...
- Final stage adders such as Brent-Kung, Ladner-Fischer, Carry-lookahead...

We measure the total time for:

- Proof events for each adder module in Step 1
- Proof event for the multiplier module in Step 2

Experiments

Our method is tested for various integer multipliers with:

- Signed/unsigned simple or Booth Encoded partial products.
- Summation trees such as Wallace-tree, Dadda-tree, 4:2 compressor trees...
- Final stage adders such as Brent-Kung, Ladner-Fischer, Carry-lookahead...

We measure the total time for:

- Proof events for each adder module in Step 1
- Proof event for the multiplier module in Step 2

Experiments (cntd.)

Table: Average proof-time results in seconds with success rate for various multiplier designs

Size	AM[i]		DK[ii]		Our Tool	
	Success	Time	Success	Time	Success	Time
64x64	8/13	89.75	24/26	19.04	26/26	2.5
128x128	4/8	577.5	14/16	305	16/16	6.19
256x256	2/7	15451	12/14	4468	14/14	22.57
512x512	0/6	-	8/12	1602	12/12	152
1024x1024	0/4	-	8/8	13906	8/8	355

i. Mahzoon, A., Große, D., Drechsler, R.: RevSCA: Using Reverse Engineering to Bring Light into Backward Rewriting for Big and Dirty Multipliers. DAC '19

ii. Kaufmann, D., Biere, A., Kauers, M.: Incremental Column-wise Verification of Arithmetic Circuits Using Computer Algebra. FMCAD '19

- Success rate={verified}/{all benchmarks}. The other tools failed either due to time-out or some program error. Failed proofs are not included in the averages.
- For the 8 benchmark types that DK[ii] could finish without time-out, their results are {**6, 40, 222, 1602, 13906**} and ours are {**1, 3, 11, 68, 355**}.

Experiments (cntd.)

Table: Average proof-time results in seconds with success rate for various multiplier designs

Size	*AM* [i]		*DK* [ii]		Our Tool	
	Success	Time	Success	Time	Success	Time
64x64	8/13	89.75	24/26	19.04	26/26	2.5
128x128	4/8	577.5	14/16	305	16/16	6.19
256x256	2/7	15451	12/14	4468	14/14	22.57
512x512	0/6	-	8/12	1602	12/12	152
1024x1024	0/4	-	8/8	13906	8/8	355

i. Mahzoon, A., Große, D., Drechsler, R.: RevSCA: Using Reverse Engineering to Bring Light into Backward Rewriting for Big and Dirty Multipliers. DAC '19

ii. Kaufmann, D., Biere, A., Kauers, M.: Incremental Column-wise Verification of Arithmetic Circuits Using Computer Algebra. FMCAD '19

- Success rate={verified}/{all benchmarks}. The other tools failed either due to time-out or some program error. Failed proofs are not included in the averages.
- For the 8 benchmark types that DK[ii] could finish without time-out, their results are {**6, 40, 222, 1602, 13906**} and ours are {**1, 3, 11, 68, 355**}.

Experiments (cntd.)

Table: Average proof-time results in seconds with success rate for various multiplier designs

Size	*AM*[i]		DK[ii]		Our Tool	
	Success	Time	Success	Time	Success	Time
64x64	8/13	89.75	24/26	19.04	26/26	2.5
128x128	4/8	577.5	14/16	305	16/16	6.19
256x256	2/7	15451	12/14	4468	14/14	22.57
512x512	0/6	-	8/12	1602	12/12	152
1024x1024	0/4	-	8/8	13906	8/8	355

i. Mahzoon, A., Große, D., Drechsler, R.: RevSCA: Using Reverse Engineering to Bring Light into Backward Rewriting for Big and Dirty Multipliers. DAC '19

ii. Kaufmann, D., Biere, A., Kauers, M.: Incremental Column-wise Verification of Arithmetic Circuits Using Computer Algebra. FMCAD '19

- Success rate={verified}/{all benchmarks}. The other tools failed either due to time-out or some program error. Failed proofs are not included in the averages.
- For the 8 benchmark types that DK[ii] could finish without time-out, their results are {**6, 40, 222, 1602, 13906**} and ours are {**1, 3, 11, 68, 355**}.

Experiments (cntd.)

Table: Average proof-time results in seconds with success rate for various multiplier designs

Size	AM[i]		*DK* [ii]		Our Tool	
	Success	Time	Success	Time	Success	Time
64x64	8/13	89.75	24/26	19.04	26/26	2.5
128x128	4/8	577.5	14/16	305	16/16	6.19
256x256	2/7	15451	12/14	4468	14/14	22.57
512x512	0/6	-	8/12	1602	12/12	152
1024x1024	0/4	-	8/8	13906	8/8	355

i. Mahzoon, A., Große, D., Drechsler, R.: RevSCA: Using Reverse Engineering to Bring Light into Backward Rewriting for Big and Dirty Multipliers. DAC '19

ii. Kaufmann, D., Biere, A., Kauers, M.: Incremental Column-wise Verification of Arithmetic Circuits Using Computer Algebra. FMCAD '19

- Success rate={verified}/{all benchmarks}. The other tools failed either due to time-out or some program error. Failed proofs are not included in the averages.
- For the 8 benchmark types that DK[ii] could finish without time-out, their results are {**6, 40, 222, 1602, 13906**} and ours are {**1, 3, 11, 68, 355**}.

Experiments (cntd.)

Table: Average proof-time results in seconds with success rate for various multiplier designs

Size	AM[i]		DK[ii]		Our Tool	
	Success	Time	Success	Time	Success	Time
64x64	8/13	89.75	24/26	19.04	26/26	2.5
128x128	4/8	577.5	14/16	305	16/16	6.19
256x256	2/7	15451	12/14	4468	14/14	22.57
512x512	0/6	-	8/12	1602	12/12	152
1024x1024	0/4	-	8/8	13906	8/8	355

i. Mahzoon, A., Große, D., Drechsler, R.: RevSCA: Using Reverse Engineering to Bring Light into Backward Rewriting for Big and Dirty Multipliers. DAC '19

ii. Kaufmann, D., Biere, A., Kauers, M.: Incremental Column-wise Verification of Arithmetic Circuits Using Computer Algebra. FMCAD '19

- Success rate={verified}/{all benchmarks}. The other tools failed either due to time-out or some program error. Failed proofs are not included in the averages.
- For the 8 benchmark types that DK[ii] could finish without time-out, their results are {**6, 40, 222, 1602, 13906**} and ours are {**1, 3, 11, 68, 355**}.

Experiments (cntd.)

Table: Average proof-time results in seconds with success rate for various multiplier designs

Size	AM[i]		DK[ii]		Our Tool	
	Success	Time	Success	Time	Success	Time
64x64	8/13	89.75	24/26	19.04	26/26	2.5
128x128	4/8	577.5	14/16	305	16/16	6.19
256x256	2/7	15451	12/14	4468	14/14	22.57
512x512	0/6	-	8/12	1602	12/12	152
1024x1024	0/4	-	8/8	13906	8/8	355

i. Mahzoon, A., Große, D., Drechsler, R.: RevSCA: Using Reverse Engineering to Bring Light into Backward Rewriting for Big and Dirty Multipliers. DAC '19

ii. Kaufmann, D., Biere, A., Kauers, M.: Incremental Column-wise Verification of Arithmetic Circuits Using Computer Algebra. FMCAD '19

- Success rate=$\{$verified$\}/\{$all benchmarks$\}$. The other tools failed either due to time-out or some program error. Failed proofs are not included in the averages.
- For the 8 benchmark types that DK[ii] could finish without time-out, their results are $\{\mathbf{6, 40, 222, 1602, 13906}\}$ and ours are $\{\mathbf{1, 3, 11, 68, 355}\}$.

Experiments (cntd.)

Table: Average proof-time results in seconds with success rate for various multiplier designs

Size	AM[i]		DK[ii]		Our Tool	
	Success	Time	Success	Time	Success	Time
64x64	8/13	89.75	24/26	19.04	26/26	2.5
128x128	4/8	577.5	14/16	305	16/16	6.19
256x256	2/7	15451	12/14	4468	14/14	22.57
512x512	0/6	-	8/12	1602	12/12	152
1024x1024	0/4	-	8/8	13906	8/8	355

i. Mahzoon, A., Große, D., Drechsler, R.: RevSCA: Using Reverse Engineering to Bring Light into Backward Rewriting for Big and Dirty Multipliers. DAC '19

ii. Kaufmann, D., Biere, A., Kauers, M.: Incremental Column-wise Verification of Arithmetic Circuits Using Computer Algebra. FMCAD '19

- Success rate={verified}/{all benchmarks}. The other tools failed either due to time-out or some program error. Failed proofs are not included in the averages.
- For the 8 benchmark types that DK[ii] could finish without time-out, their results are {**6, 40, 222, 1602, 13906**} and ours are {**1, 3, 11, 68, 355**}.

Experiments (cntd.)

Table: Average proof-time results in seconds with success rate for various multiplier designs

Size	AM[i]		DK[ii]		Our Tool	
	Success	Time	Success	Time	Success	Time
64x64	8/13	89.75	24/26	19.04	26/26	2.5
128x128	4/8	577.5	14/16	305	16/16	6.19
256x256	2/7	15451	12/14	4468	14/14	22.57
512x512	0/6	-	8/12	1602	12/12	152
1024x1024	0/4	-	8/8	13906	8/8	355

i. Mahzoon, A., Große, D., Drechsler, R.: RevSCA: Using Reverse Engineering to Bring Light into Backward Rewriting for Big and Dirty Multipliers. DAC '19

ii. Kaufmann, D., Biere, A., Kauers, M.: Incremental Column-wise Verification of Arithmetic Circuits Using Computer Algebra. FMCAD '19

- Success rate={verified}/{all benchmarks}. The other tools failed either due to time-out or some program error. Failed proofs are not included in the averages.
- For the 8 benchmark types that DK[ii] could finish without time-out, their results are {**6, 40, 222, 1602, 13906**} and ours are {**1, 3, 11, 68, 355**}.

Experiments (cntd.)

Table: Average proof-time results in seconds with success rate for various multiplier designs

Size	AM[i]		DK[ii]		Our Tool	
	Success	Time	Success	Time	Success	Time
64x64	8/13	89.75	24/26	19.04	26/26	2.5
128x128	4/8	577.5	14/16	305	16/16	6.19
256x256	2/7	15451	12/14	4468	14/14	22.57
512x512	0/6	-	8/12	1602	12/12	152
1024x1024	0/4	-	8/8	13906	8/8	355

i. Mahzoon, A., Große, D., Drechsler, R.: RevSCA: Using Reverse Engineering to Bring Light into Backward Rewriting for Big and Dirty Multipliers. DAC '19

ii. Kaufmann, D., Biere, A., Kauers, M.: Incremental Column-wise Verification of Arithmetic Circuits Using Computer Algebra. FMCAD '19

- Success rate={verified}/{all benchmarks}. The other tools failed either due to time-out or some program error. Failed proofs are not included in the averages.
- For the 8 benchmark types that DK[ii] could finish without time-out, their results are {**6, 40, 222, 1602, 13906**} and ours are {**1, 3, 11, 68, 355**}.

Conclusions

- We have proposed an efficient method to verify integer multipliers
- We work with a wide-range of different designs.
- Our method is proved correct using ACL2
- Future work:
  - Create counterexamples
  - Support completely flattened designs

Conclusions

- We have proposed an efficient method to verify integer multipliers
- We work with a wide-range of different designs.
- Our method is proved correct using ACL2
- Future work:
  - Create counterexamples
  - Support completely flattened designs

Conclusions

- We have proposed an efficient method to verify integer multipliers
- We work with a wide-range of different designs.
- Our method is proved correct using ACL2
- Future work:
  - Create counterexamples
  - Support completely flattened designs

Conclusions

- We have proposed an efficient method to verify integer multipliers
- We work with a wide-range of different designs.
- Our method is proved correct using ACL2
- Future work:
  - Create counterexamples
  - Support completely flattened designs

The End