Sound and Automated Verification of Real-World RTL Multipliers

Mertcan Temel[1,2]
mert@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

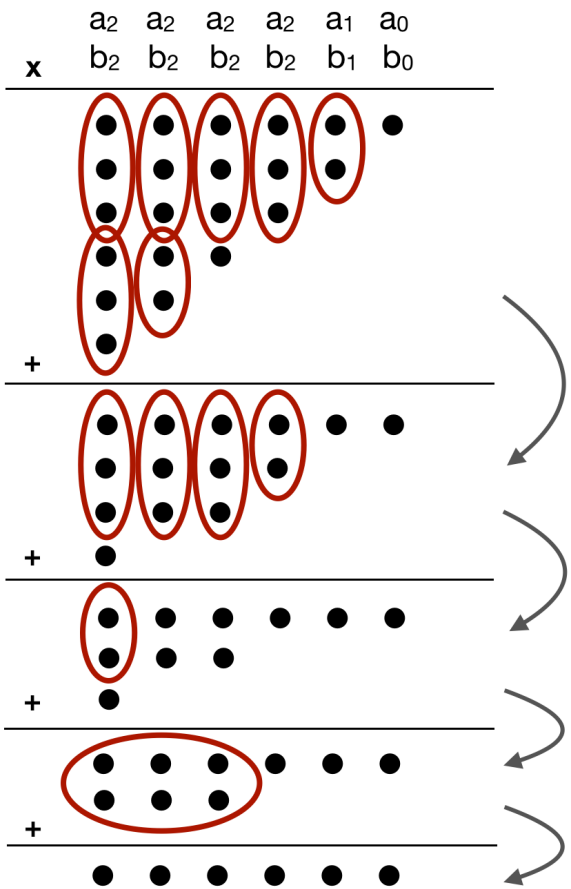
October 19-22, 2021 (FMCAD 2021)

Introduction

- Integer multipliers are ubiquitous components.
- Wallace-tree and Booth Encoding are complex algorithms to design efficient integer multipliers.
- Verification of these multipliers is a difficult problem.
- This work proposes a more efficient, rewrite-based method that is:
  - widely applicable (tested for 250+ benchmarks),
  - scalable (1024x1024-bit multipliers proved in 5 minutes),
  - provably correct (multiplier verification procedure is verified using ACL2)

Overview

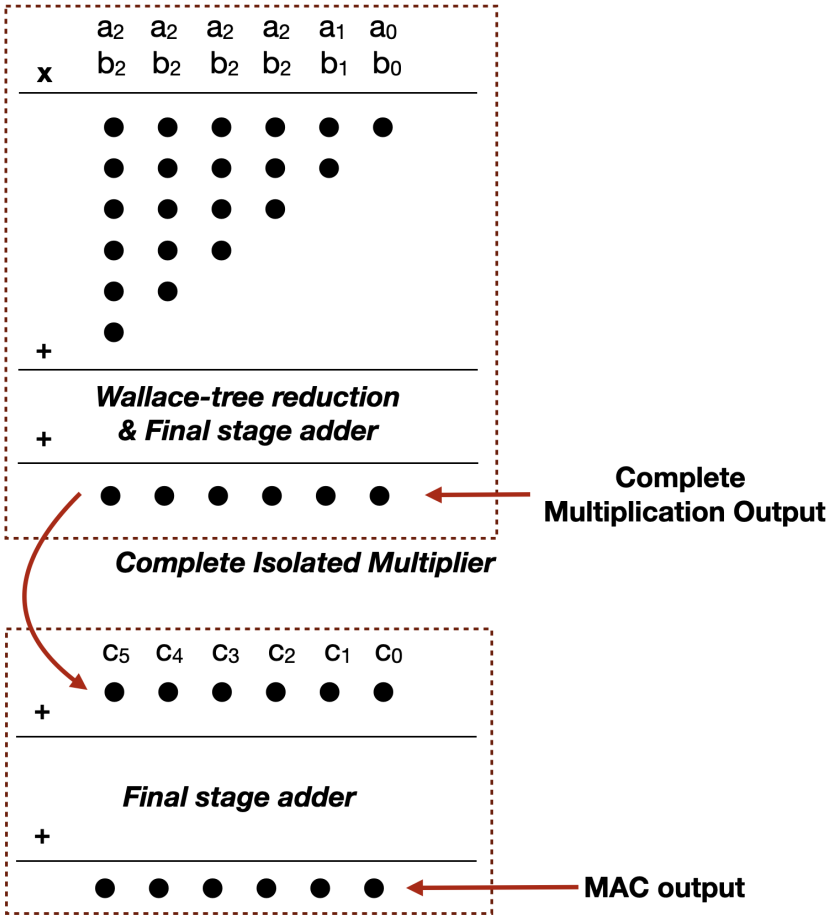
Review: Integer Multipliers



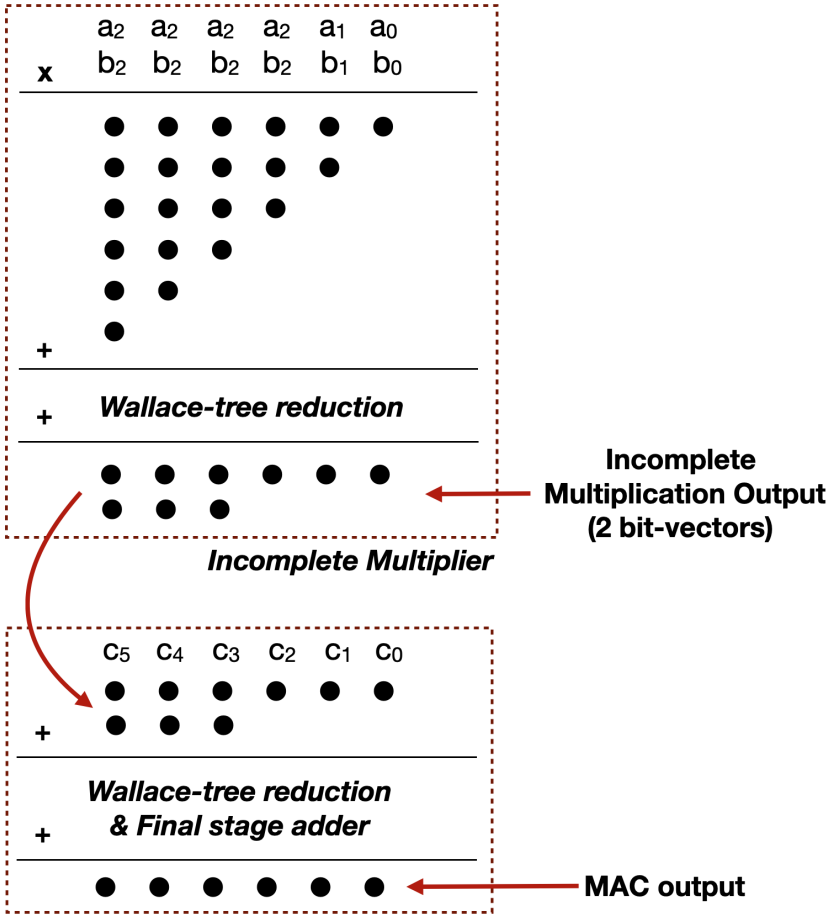
3x3-bit signed multiplication with Wallace-tree

- Integer multipliers have two stages:
  1. Partial Product Generation (e.g., Simple, Baugh-Wooley, Booth Encoding)
  2. Partial Product Summation (e.g., Array, Wallace-tree, Dadda-tree)
- Booth encoding and Wallace-tree are preferred methods for their gate-delay benefits but they are more difficult to verify.
- Designs may involve: rounding, saturation, truncation, right-shifting.
- Some use cases: multiply-accumulate, dot-product, floating-point multiplication.

Review: Integer Multipliers (cntd.)



A Multiply-Accumulate (MAC) implementation with large gate delay



Another way to implement a MAC module with less gate delay

Related Work

- Computer algebra based methods have shown a lot of progress. However:
  - Only tested for isolated multipliers (or for only simple architectures)
  - Problems verifying truncated multipliers
  - Limited discussion on the correctness of the verification tools themselves
- Industrial designs are verified mostly manually.

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_64x64_mult>)
                  (truncate 128
                            (* (signext 64 a)
                               (signext 64 b))))))
```

- ACL2 is an interactive and automated theorem proving system.
- RTL-level hierarchical 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.
- Specification (right-hand side) can be modified by the user to suit the design

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_64x64_mult>)
                  (truncate 128
                            (* (signext 64 a)
                               (signext 64 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	-	-	$c(a_0 + a_1)$	$s(a_0 + a_1)$
Full-adder	-	-	$c(a_0 + a_1 + a_2)$	$s(a_0 + a_1 + a_2)$
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 128
                            (* (signext 64 a)
                               (signext 64 b))))))
```

Both LHS and RHS should be rewritten to the same form. For example:

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: 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 1 $\forall x, y \in \mathbb{Z}\ s(s(x) + y) = s(x + y)$

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 2: Simplify Summation Trees (cntd.)

The current term:

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

Some c instances can be rewritten with the below lemma.

Lemma 2 $\forall x, y \in \mathbb{Z}\ c(s(x) + y) = c(x + y) - c(x)$

When this lemma is 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 target final form. This rewriting method works for much larger terms as well.

Booth encoding might yield complex partial product bits. They are simplified with another set of lemmas that work together with Lemmas 1-2. They are omitted here.

Experiments

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	34/34	44	34/34	18.5	34/34	1.7
128x128	14/16	510.5	14/16	238	16/16	2.5
256x256	4/16	8918	14/16	3855	16/16	10.1
512x512	0/12	-	8/12	1602	12/12	47
1024x1024	0/12	-	8/12	13906	12/12	246

- Success rate=$\{$verified$\}/\{$all benchmarks$\}$. Unsuccessful results due to time-out. Only successful results are included in averages.
- Other tests include: multiply-accumulate, dot-product, multipliers with control logic, truncated or right-shifted outputs, rounding, multipliers with arbitrary operand sizes. Also verified private real-world designs in Centaur Technology. Total of 250+ example designs.

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

Conclusions

- An efficient method to verify integer multipliers. Compared to the other state-of-the-art tools:
  - it can verify large multipliers in a much shorter amount of time,
  - it has wider applicability,
  - used in real-world designs,
  - and it is verified
- A distinctive verification approach for its scale.