Mertcan Temel

Formal Verification of Booth Radix-8 and Radix-16 Multipliers

DATE 2024 (Design, Automation and Test in Europe)

intel®

Introduction

- Integer multipliers are very common -> many design optimizations for best performance
  - E.g., Booth radix-8 or radix-16 for lower power consumption
- Formal verification is necessary but also very difficult.
  - Commercial designs were previously verified with heavily manual methods
  - We aim to make multiplier verification faster and more automatic
  - S-C-Rewriting (the method) as implemented in VeSCMul (the tool) has been successful in verifying commercial multiplier designs
- This talk goes over how S-C-Rewriting is extended to support radix-8 and radix-16 multipliers

Booth Encoding Summary

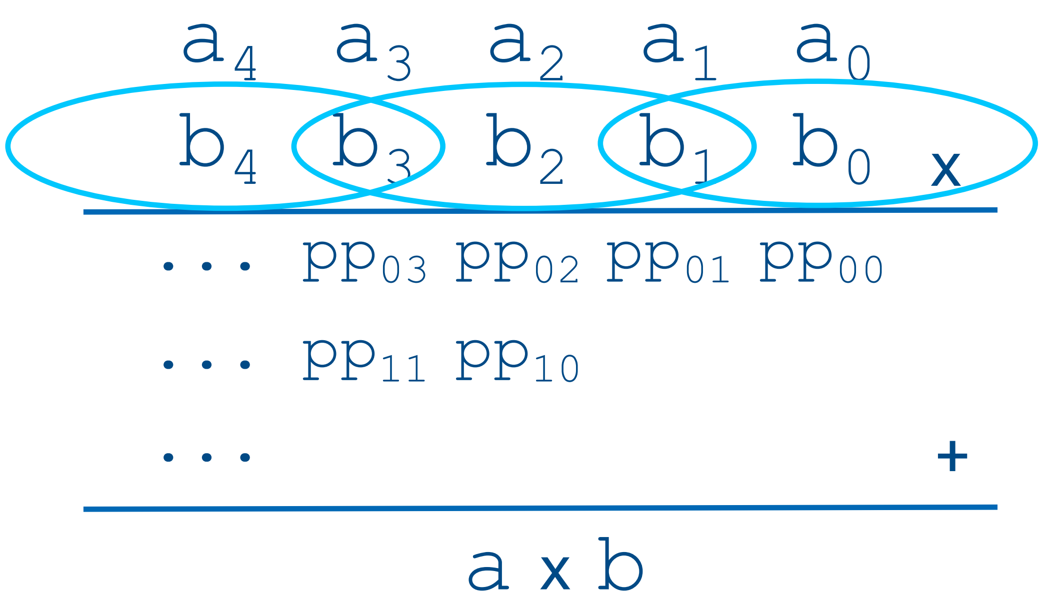
No Booth encoding case (aka simple partial products): Simply lay out products of input bits to be summed for the result.

$$
\begin{array}{rccccccc}
 & & a_4 & a_3 & a_2 & a_1 & a_0 & \\
 & & b_4 & b_3 & b_2 & b_1 & b_0 & \times \\
\hline
 & \ldots & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 & & \\
 & \ldots & a_2b_1 & a_1b_1 & a_0b_1 & & & \\
 & \ldots & a_1b_2 & a_0b_2 & & & & \\
 & \ldots & a_0b_3 & & & & & \\
 & \ldots & & & & & & + \\
\hline
 & & & a \times b & & & &
\end{array}
$$

← Easier to verify but not a good design choice

Booth Encoding Summary

In Booth encoding, multiplies of the multiplicand operand is selected by multiplier operand bits for each partial product row.



Radix-4: select 3 bits of multiplier at a time to calculate pp logic

Smaller summation tree, but more complex pp logic.

Booth Encoding Summary

Booth Encoding can be various radices. Radix-2 selects 2-bits of multiplier; radix-4 selects 3; radix-8 selects 4…

- Radix-2, 4: coefficients of multiplicands are all power of 2

  For example: {-2x, -1x, 0, 1x, 2x} for radix-4

- Radix-8,16: some coefficients are not power of 2

  For example: {-4x, -3x, -2x, -1x, 0, 1x, 2x, 3x, 4x} for radix-8

  -> vector addition in pp logic

This affects state-of-the-art verification procedures.

S-C-Rewriting - Recap

- A term-rewriting based method targeting RTL multiplier designs
- Implemented on ACL2 as part of the VeSCMul tool. It is fully verified.
- Very fast. E.g., 64x64-bit multipliers in seconds, 1024x1024 radix-4 in minutes
- Very comprehensive. Supports variations used in commercial designs:
  - Multiply-add/subtract, dot product operations and others
  - Custom operand sizes (17x34-bit multiplication)
  - Output truncation/right-shifting

S-C-Rewriting - Recap

- Partial product logic is rewritten with *algebraic rewriting*:

  Lemma 1. $\forall x \in \{0,1\}\ \overline{x} \rightarrow 1 - x$ Lemma 2. $\forall\, \mathrm{x}, \mathrm{y} \in \{0,1\}\ \mathrm{x} \wedge \mathrm{y} \rightarrow \mathrm{xy}$

  - E.g., $\overline{x} \wedge (y \wedge \overline{z})$ is rewritten to $-xy + xyz + y - yz$.
  - This alone is too expensive for radix-8 and radix-16 multipliers.
- Remainder parts (adders) are rewritten to the **s** and **c** functions.
  - $s(x) = mod_2(x), \ \ c(x) = \lfloor {}^{x}/_{2} \rfloor$
  - $\forall x, y, z \in \{0,1\}\ fulladder(x, y, z) \rightarrow \{carry: c(x + y + z),\ sum: s(x + y + z)\}$
  - Resulting s and c terms are simplified with a custom rewriting methodology

Improvements to S-C-Rewriting

We have made 3 distinct improvements for scalable verification of high-radix multipliers:

A. No Algebraic Rewriting for Addition Logic in Partial Products
B. A Shortcut Rewrite Rule
C. Dynamically Learn Pattern Reductions

Improvement A: Rewriting Addition Logic in PP

Addition in PP for radix-8+ causes scalability issues in the old method.

Solution: Rewrite the adders in PP to the s and c functions

-> Now, we start seeing some good results:

	Radix-8					Radix-16				
	8x8	**16**	**32**	**64**	**128**	**8x8**	**16**	**32**	**64**	**128**
Before	.2s	7.8s	St. Ov	St. Ov	St. Ov	2.3s	St. Ov	St. Ov	St. Ov	St. Ov
After Impr. A	.1s	.3s	3.6s	145s	81m	.3s	2.5s	104s	74m	TO

St. Ov: Stack overflow. TO: time-out.
Additional experimental results for radix-4 is available on the paper.

Improvement B: A Shortcut Rewrite Rule

Improvement A creates new term patterns.

Solution: a new shortcut rewrite rule

$$\forall x, y \in Z \; c(-s(x) + y) \rightarrow c(x + y) + c(x) - x$$

-> Larger multipliers now scale:

	Radix-8					Radix-16				
	8x8	**16**	**32**	**64**	**128**	**8x8**	**16**	**32**	**64**	**128**
Before	.2s	7.8s	St. Ov	St. Ov	St. Ov	2.3s	St. Ov	St. Ov	St. Ov	St. Ov
After Impr. A	.1s	.3s	3.6s	145s	81m	.3s	2.5s	104s	74m	TO
After Impr. A&B	.1s	.3s	1.3s	5s	20.4s	.3s	1.6s	7.2s	31s	128s

Improvement C: Learn Pattern Reductions

System performs the same pattern reduction for different variables.

Solution: Dynamically learn pattern reductions for gate groups.

-> Up to ~4x further improvement:

	Radix-8					Radix-16				
	8x8	**16**	**32**	**64**	**128**	**8x8**	**16**	**32**	**64**	**128**
Before	.2s	7.8s	St. Ov	St. Ov	St. Ov	2.3s	St. Ov	St. Ov	St. Ov	St. Ov
After Impr. A	.1s	.3s	3.6s	145s	81m	.3s	2.5s	104s	74m	TO
After Impr. A&B	.1s	.3s	1.3s	5s	20.4s	.3s	1.6s	7.2s	31s	128s
After Impr. A&B&C	.1s	.2s	.6s	2.1s	8.6s	.2s	.5s	1.7s	7.5s	33s

Conclusion

- Multiplier verification is an important step in a processor design project
- Various optimizations (e.g., radix-8, radix-16) might be used in commercial designs
- Automatic and fast verification of multipliers is valuable
- Could not scalably verify high-radix multipliers before. Now we can.
- With the 3 improvements, S-C-Rewriting is very fast and automatic.
- S-C-Rewriting saves us time by quickly verifying commercial multipliers