Method for Multiplier Verification Employing Boolean Equivalence Checking and Arithmetic Bit Level Description

U. Krautz\textsuperscript{1}, M. Wedler\textsuperscript{1}, W. Kunz\textsuperscript{1} & K. Weber\textsuperscript{2}, C. Jacobi\textsuperscript{2}, M. Pflanz\textsuperscript{2}

\textsuperscript{1}University of Kaiserslautern - Germany
\textsuperscript{2}IBM Forschung und Entwicklungs GmbH - Germany

13th Asia and South Pacific Design Automation Conference, 2008
Outline

1 Motivation
   - The Basic Problem
   - Other Approaches

2 Our Methodology
   - Excursion: Multiplier Design
   - ARDL
   - Overview
   - Arithmetic Proof
   - Equivalence Check

3 Our Results/Contribution
   - Main Results
Outline

1 Motivation
   - The Basic Problem
   - Other Approaches

2 Our Methodology
   - Excursion: Multiplier Design
   - ARDL
   - Overview
   - Arithmetic Proof
   - Equivalence Check

3 Our Results/Contribution
   - Main Results
formal property checking of significant importance

- arithmetic circuits still “show stoppers”
- no universal framework for arithmetics, instead special “engineered” solutions
- useful for highly regular designs
- limited in case of full custom logic designs
- multipliers particular hard to verify
  - hardware multipliers common in processors
  - hard to generate compact canonical representation from bit level
  - for verification often equivalence check against reference (reference and design have to share large structural similarities)
General Arithmetic Circuit Verification

- formal property checking of significant importance
- arithmetic circuits still “show stoppers”
- no universal framework for arithmetics, instead special “engineered” solutions
- useful for highly regular designs
- limited in case of full custom logic designs
- multipliers particular hard to verify
  - hardware multipliers common in processors
  - hard to generate compact canonical representation from bit level
  - for verification often equivalence check against reference (reference and design have to share large structural similarities)
General Arithmetic Circuit Verification

- formal property checking of significant importance
- arithmetic circuits still “show stoppers”
- no universal framework for arithmetics, instead special “engineered” solutions
- useful for highly regular designs
- limited in case of full custom logic designs
- multipliers particular hard to verify
  - hardware multipliers common in processors
  - hard to generate compact canonical representation from bit level
  - for verification often equivalence check against reference (reference and design have to share large structural similarities)
General Arithmetic Circuit Verification

- formal property checking of significant importance
- arithmetic circuits still “show stoppers”
- no universal framework for arithmetics, instead special “engineered” solutions
- useful for highly regular designs
- limited in case of full custom logic designs
- multipliers particular hard to verify
  - hardware multipliers common in processors
  - hard to generate compact canonical representation from bit level
  - for verification often equivalence check against reference (reference and design have to share large structural similarities)
General Arithmetic Circuit Verification

- formal property checking of significant importance
- arithmetic circuits still “show stoppers”
- no universal framework for arithmetics, instead special “engineered” solutions
- useful for highly regular designs
- limited in case of full custom logic designs
- multipliers particular hard to verify
  - hardware multipliers common in processors
  - hard to generate compact canonical representation from bit level
  - for verification often equivalence check against reference (reference and design have to share large structural similarities)
General Arithmetic Circuit Verification

- formal property checking of significant importance
- arithmetic circuits still “show stoppers”
- no universal framework for arithmetics, instead special “engineered” solutions
- useful for highly regular designs
- limited in case of full custom logic designs
- multipliers particular hard to verify
  - hardware multipliers common in processors
  - hard to generate compact canonical representation from bit level
  - for verification often equivalence check against reference (reference and design have to share large structural similarities)
Outline

1 Motivation
   • The Basic Problem
   • Other Approaches

2 Our Methodology
   • Excursion: Multiplier Design
   • ARDL
   • Overview
   • Arithmetic Proof
   • Equivalence Check

3 Our Results/Contribution
   • Main Results
Previous Work

- Binary Moment Diagrams (*BMD) [Bryant, Cheng, Hamaguchi]
  - lack of robustness
- functional decomposition [Chang, Cheng, Fujita, Chen, Aagaard, Seger, Kaivola, Narasimhan]
  - prove internal properties
  - compose global proof of sub-goals
  - manual decomposition
  - non-trivial mapping of lowest proof level to design
- comparison of reference and design based on 1bit-adder network [Stoffel, Wedler]
  - extraction of adder network from design and reference (exponential number of possibilities)
  - equivalence proven by simple calculus
Previous Work

- Binary Moment Diagrams (*BMD) [Bryant, Cheng, Hamaguchi]
  - lack of robustness

- functional decomposition [Chang, Cheng, Fujita, Chen, Aagaard, Seger, Kaivola, Narasimhan]
  - prove internal properties
  - compose global proof of sub-goals
  - manual decomposition
  - non-trivial mapping of lowest proof level to design

- comparison of reference and design based on 1bit-adder network [Stoffel, Wedler]
  - extraction of adder network from design and reference (exponential number of possibilities)
  - equivalence proven by simple calculus
Previous Work

- Binary Moment Diagrams (*BMD) [Bryant, Cheng, Hamaguchi]
  - lack of robustness
- functional decomposition [Chang, Cheng, Fujita, Chen, Aagaard, Seger, Kaivola, Narasimhan]
  - prove internal properties
  - compose global proof of sub-goals
  - manual decomposition
  - non-trivial mapping of lowest proof level to design
- comparison of reference and design based on 1bit-adder network [Stoffel, Wedler]
  - extraction of adder network from design and reference (exponential number of possibilities)
  - equivalence proven by simple calculus
Outline

1 Motivation
   - The Basic Problem
   - Other Approaches

2 Our Methodology
   - Excursion: Multiplier Design
   - ARDL
   - Overview
   - Arithmetic Proof
   - Equivalence Check

3 Our Results/Contribution
   - Main Results
A Simple Multiplier

**Algorithm**

- integer multiplication
  
  \[(a_0, \ldots, a_n) \cdot (b_0, \ldots, b_m) = p\]

- basic multiplication (grade school algorithm):

  \[
  \begin{align*}
  (a_0, \ldots, a_n) \cdot 2^0 & \cdot b_0 \\
  + (a_0, \ldots, a_n) \cdot 2^1 & \cdot b_1 \\
  \vdots & \vdots \\
  + (a_0, \ldots, a_n) \cdot 2^{m-1} & \cdot b_{m-1} \\
  + (a_0, \ldots, a_n) \cdot 2^m & \cdot b_m
  \end{align*}
  \]

**Structure**

- Partial product generation
- Addition network
- Product

U. Krautz et al.  Multiplier Verification
Full Custom Multipliers

- custom multipliers @IBM developed on bit-level
- various optimizations
  - no half-adder instances (just “AND”, “XOR”-gates)
  - full-adders spread across cycles
  - no booth-encoder instances (just shifter, multiplexer)
  - constant bits in adder-tree
  - hot-one/ hot-two representation
- no word level information available (hard to extract)
Outline

1 Motivation
   - The Basic Problem
   - Other Approaches

2 Our Methodology
   - Excursion: Multiplier Design
   - ARDL
     - Overview
     - Arithmetic Proof
     - Equivalence Check

3 Our Results/Contribution
   - Main Results
Arithmetic Reference Description Language

- utilized in verification
- specify multiplier on word level:
  - arithmetic by functions
  - structure by interconnection between functions
- syntax close to HD languages (Verilog / VHDL)
  - used by designer, not verification engineer
- developed at beginning of design process
  - formalization of typical considerations before implementing a design
variables {
    a: in (0 to 2);
    b: in (0 to 2);
    ...
}
pp_def{
    ppb(0) <= gen_pp(a(0 to 2), booth22(0 & 0 & b(0)));
    ppb(1) <= gen_pp(a(0 to 2), booth22(b(0 to 2)));
    ...
}
tree_def{
    s1a <= sum32 (ppb(0), ppb(1), ppb(2));
    c1a <= carry32 (ppb(0), ppb(1), ppb(2));
    sum <= s1a;
    carry <= c1a
    ...
}
Outline

1 Motivation
   - The Basic Problem
   - Other Approaches

2 Our Methodology
   - Excursion: Multiplier Design
   - ARDL
   - Overview
   - Arithmetic Proof
   - Equivalence Check

3 Our Results/Contribution
   - Main Results
Approach Overview

- **Fix Reference**
- Extract adder network
- **Reference (adder network)**
- Is correct arithmetic function?
- Yes
  - Design correct
- No
  - **Reference (adder network)**
  - Fix Reference

- **Convert to gate netlist**
- **Reference (gate netlist)**
- Is equivalent?
- Yes
  - Design correct
- No
  - **Design (gate netlist)**
  - Fix Design

- **ARDL**
- **Overview**
- **Arithmetic Proof**
- **Equivalence Check**
Outline

1 Motivation
   - The Basic Problem
   - Other Approaches

2 Our Methodology
   - Excursion: Multiplier Design
   - ARDL
   - Overview
   - Arithmetic Proof
   - Equivalence Check

3 Our Results/Contribution
   - Main Results
Overview Arithmetic Proof

1. Reference (ARDL)
2. Extract adder network
3. Convert to gate netlist
4. Design (gate netlist)
5. Fix Design
6. Reference (adder network)
7. is equivalent?
8. Design correct
9. correct arithmetic function?
10. yes
11. yes
12. no
13. no
ARDL to Structure

**structural view**

- input: ARDL of multiplier design (given by designer)

**functional view**

- \( a = (a_0, \ldots, a_n) \)
- \( b = (b_0, \ldots, b_m) \)
- \( pp_j = a \cdot B_j \)
- \( B_j = -2ab_{j+1} + ab_j + ab_{j-1} \)
- \( prod = \sum_{j=0}^{m} pp_j \)

- arithm. functions derived from ARDL structure (automatically)
  - Booth encoding
  - adder network
**Structure to Bit Arithmetic**

**functional view**

\[
a = (a_0, \ldots, a_n), \quad b = (b_0, \ldots, b_m)
\]

\[
pp_j = a \cdot B_j, \\
B_j = -2ab_{j+1} + ab_j + ab_{j-1}
\]

\[
\text{prod} = \sum_{j=0}^{m} pp_j
\]

- arithm. functions derived from ARDL structure (automatically)
- Booth encoding
- adder network

**bit arithmetic**

\[
\begin{array}{ccccccc}
0 & 0 & 0 & 0 & -2a_0b_2 \\
0 & a_0b_1 & a_1b_1 & a_2b_1 \\
a_1b_0 & a_0b_2 & a_1b_2 & a_2b_2 \\
\end{array}
\]

- transformation to basic multiplier definition (automatically)
- successful/unsuccessful → correct/incorrect arithmetic
Example Booth encoded 3bit multiplier

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$-2a_0b_2$</td>
<td>$-2a_1b_2$</td>
<td>$-2a_2b_2$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>$a_0b_1$</td>
<td>$a_1b_1$</td>
<td>$a_2b_1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>$a_0b_2$</td>
<td>$a_1b_2$</td>
<td>$a_2b_2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>$-2a_0b_0$</td>
<td>$-2a_1b_0$</td>
<td>$-2a_2b_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$a_0b_0$</td>
<td>$a_1b_0$</td>
<td>$a_2b_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

$p_{-1}$ $p_0$ $p_1$ $p_2$ $p_3$ $p_4$
Example Booth encoded 3bit multiplier

after transformation

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -a_0 b_2 & -a_1 b_2 & -a_2 b_2 & 0 \\
0 & 0 & a_0 b_1 & a_1 b_1 & a_2 b_1 & 0 & 0 & 0 \\
0 & 0 & 0 & 2a_0 b_2 & 2a_1 b_2 & 2a_2 b_2 & 0 & 0 \\
0 & -a_0 b_0 & -a_1 b_0 & -a_2 b_0 & 0 & 0 & 0 & 0 \\
0 & 2a_0 b_0 & 2a_1 b_0 & 2a_2 b_0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[ p_{-1} \quad p_0 \quad p_1 \quad p_2 \quad p_3 \quad p_4 \]
Example Booth encoded 3bit multiplier

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>p_{-1}</td>
<td>p_0</td>
<td>p_1</td>
<td>p_2</td>
<td>p_3</td>
<td>p_4</td>
<td></td>
</tr>
</tbody>
</table>
Outline

1 Motivation
   - The Basic Problem
   - Other Approaches

2 Our Methodology
   - Excursion: Multiplier Design
   - ARDL
   - Overview
   - Arithmetic Proof
   - Equivalence Check

3 Our Results/Contribution
   - Main Results
Overview - Equivalence Check

- **Reference (ARDL)**
  - Extract adder network
  - Correct arithmetic function?
    - no
    - yes
      - Design correct
  - Design (gate netlist)
    - is equivalent?
      - no
      - yes

- **Fix Reference**

- **Fix Design**

- **Extract adder network**

- **Convert to gate netlist**

- **U.Krautz et al. Multiplier Verification 23**
Equivalence Check

- ARDL structure translated into gate-list reference
- Functions synthesized into pre-defined blocks (Booth-encoder, adder, ...)
- Equivalence checked of reference against design (standard SAT solver)
- Successful check requires similarities between reference and design
  - Similar inputs to adder-tree (same Booth-encoding)
  - Adder-tree topology
  - Assignments to adders (order of inputs)
- Similarities through design concept in ARDL
  - Similarities result through methodology
  - Designer’s responsibility
Motivation
Our Methodology
Our Results/Contribution
Summary

Excursion: Multiplier Design
ARDL
Overview
Arithmetic Proof
Equivalence Check

Overview - Final

- Fix Reference
- Extract adder network
- Reference (adder network)
- correct arithmetic function?
- yes
- Design correct
- no

- Convert to gate netlist
- Reference (gate netlist)

- Design (gate netlist)
- is equivalent?
- yes
- yes
- yes

- Fix Design

References:
- U.Krautz et al.
- Multiplier Verification
Scope of Method

- arithmetic circuits modeled at word-level, with easy to derive bit-level
  - data-path verification
  - arithmetic otherwise hard to obtain from gate netlist
  - flexible

- multicycle operations through unrolling
Outline

1 Motivation
   • The Basic Problem
   • Other Approaches

2 Our Methodology
   • Excursion: Multiplier Design
   • ARDL
   • Overview
   • Arithmetic Proof
   • Equivalence Check

3 Our Results/Contribution
   • Main Results
Implementation/Experiments

- prototype implementation (PERL)
- used on several industrial multipliers (IBM)
- complex instructions
  - parallel operations in wide multipliers
  - require different ARDL for each instruction

### Main Results

<table>
<thead>
<tr>
<th>Operation (operand’s bit width)</th>
<th>cpu time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AP</td>
</tr>
<tr>
<td>4x4</td>
<td>0.6s</td>
</tr>
<tr>
<td>8x8</td>
<td>1s</td>
</tr>
<tr>
<td>8x8 + 8x8 + 8x8 + 8x8</td>
<td>9s</td>
</tr>
<tr>
<td>16x16 + 16x16</td>
<td>9s</td>
</tr>
<tr>
<td>24x24</td>
<td>7s</td>
</tr>
<tr>
<td>53x53</td>
<td>8min</td>
</tr>
<tr>
<td>64x64</td>
<td>14min</td>
</tr>
</tbody>
</table>
Conclusion

- ARDL suited for binary multiplier designs (FPU, FXU, ...)
- Manual effort negligible (formalization of typical design considerations)
- Applied to complex instructions (multiply-add)
- Unsigned, signed multiplication possible
Summary

- Verification method for multipliers:
- special reference description in ARDL (Arithmetic Reference Description Language)
- ARDL reference for simple bit arithmetic check - transformation to basic multiplication
- ARDL reference for construction of gate-list representation - equivalence check against design
Questions?