Functional Verification & Abstraction of Arithmetic Circuits

Maciej Ciesielski, Cunxi Yu

University of Massachusetts, Amherst
ciesiel@ecs.umass.edu
Functional verification

- Does the circuit implement the required arithmetic function?
- *What function* does this circuit implement?

Extracting arithmetic function from gate-level implementations

- Avoid Boolean methods, bit-blasting
- Use Computer Algebra methods
Computer Algebra Approach

- Circuit specification $F_{\text{spec}}$ and implementation $B$ represented by pseudo-Boolean polynomials
  - Check if implementation $B$ satisfies specification $F_{\text{spec}}$ by reducing $F_{\text{spec}}$ modulo $B$
    
    $F_{\text{spec}} \xrightarrow{B} + r$

    - If $r = 0$, the circuit is correct
    - If $r \neq 0$, circuit may still be correct but need canonical Groebner basis (GB) to determine if $r = 0$
    - Difficult to compute, computationally complex
    - GB must include polynomials $<x^2-x>$ (for all Boolean signals $x$)

- Methods differ in ways they accomplish reduction
  - Arithmetic Bit-level (ABL) representation [Wienand’08, Pavlenko’11]
  - Also applied to Galois Fields (GF) [Kalla’14, TComp’15]
Example: 2-bit adder

- \( F_{\text{spec}} = a_0 + b_0 + 2a_1 + 2b_1 - 4r_2 - 2r_1 - r_0 \)
- \( B = \) list of polynomials describing gates
- Proof of functional correctness done by polynomial division (similar to our forward rewriting)

\[
\begin{align*}
g_1 &= r_0 - (a_0 + b_0 - 2a_0b_0) \\
g_2 &= c - (a_0b_0) \\
g_3 &= d - (a_1 + b_1 - 2a_1b_1) \\
g_4 &= r_1 - (c + d - 2cd) \\
g_5 &= f - (cd) \\
g_6 &= e - (a_1b_1) \\
g_7 &= r_2 - (e + f - ef)
\end{align*}
\]
Polynomial Division (1)

- Divide polynomial $F_{\text{spec}} = a_0 + b_0 + 2a_1 + 2b_1 - 4r_2 - 2r_1 - r_0$
  
  $$= -(a_0 + b_0 - 2a_0b_0) + r_0 + b_0 + 2a_1 + 2b_1 - 4r_2 - 2r_1 - r_0$$
  
  $$= 2a_0b_0 + 2a_1 + 2b_1 - 4r_2 - 2r_1$$

- $- 2(a_0b_0) + 2c + 2a_0b_0 + 2a_1 + 2b_1 - 4r_2 - 2r_1$
  
  $$= 2c + 2a_1 + 2b_1 - 4r_2 - 2r_1$$

- $- 2(2a_1b_1 + 2d + 2c + 2a_1 + 2b_1 - 4r_2 - 2r_1)$
  
  $$= 4a_1b_1 - 2d + 2c - 4r_2 - 2r_1$$

- $- 2(c + 2d - 2cd) + 2r_1 + 4a_1b_1 + 2d + 2c - 4r_2 - 2r_1$
  
  $$= 4cd + 4a_1b_1 - 4r_2$$

- $- 2(c + d - 2cd) + 2r_1 + 4a_1b_1 + 2d + 2c - 4r_2 - 2r_1$
  
  $$= 4cd + 4a_1b_1 - 4r_2$$

- $- 4(2a_1b_1 + 2d + 2c + 2a_1 + 2b_1 - 4r_2 - 2r_1)$
  
  $$= 4a_1b_1 - 4r_2$$

- $- 4(a_1b_1) + 4e + 4f + 4a_1b_1 - 4r_2$
  
  $$= 4e + 4f - 4r_2$$

- $- 4(e + f - ef) + 4r_2 + 4e + 4f - 4r_2$
  
  $$= 4ef$$

Non-zero residual! Is circuit correct?
Polynomial Division (2)

- Continue dividing residual polynomial \{ 4ef \}

\[
\begin{align*}
4ef &= 4e(cd) \\
&= 4(a_1 b_1)(cd) \\
&= 4(a_1 b_1)(a_0 b_0)(a_1 + b_1 - 2a_1 b_1) \\
&= 4(a_1 b_1)(a_1 + b_1 - 2a_1 b_1)(a_0 b_0) \\
&= 4(a_1 b_1 a_1 + a_1 b_1 b_1 - 2a_1 b_1 a_1 b_1)(a_0 b_0) \\
&= 4(0)(a_0 b_0) \\
&= 4ef = 0
\end{align*}
\]

- Hence \((F_{\text{spec}} \mod B) = 0\), the circuit correctly implements a 2-bit adder.
- But many dividing steps are needed.
Another way of looking at the problem:

Instead of reducing $F_{\text{spec}}$ modulo $B$, we can

- Rewrite $\text{Sig}_{\text{in}} \rightarrow \text{Sig}_{\text{out}}$ (forward rewriting), or
- Rewrite $\text{Sig}_{\text{out}} \rightarrow \text{Sig}_{\text{in}}$ (backward rewriting)
Backward Rewriting

- Replace gate output by its equation
  - Backward *symbolic simulation*
  - No residual expression $(r)$!
  - But … the expression can explode

$$f_3 = 4r_2 + 2r_1 + r_0$$
$$f_2 = 4(f + e - ef) + 2r_1 + r_0$$
$$= 4f + 4e - 4ef + 2r_1 + r_0$$

$$f_1 = 4e + 4(cd) - 4e(cd) + 2(c + d - 2cd) + r_0$$
$$= 4e + 2c + 2d + r_0 - 4ecd$$

$$f_0 = 4(a_1b_1) + 2(a_0b_0) + 2(a_1 + b_1 - 2a_1b_1) + (a_0 + b_0 - 2a_0b_0) - 4(a_1b_1)(a_0b_0)(a_1 + b_1 - 2a_1b_1)$$
$$= 2a_1 + 2b_1 + a_0 + b_0$$

It matches the specification:
\[ \rightarrow \text{circuit is correct} \]
How efficient is it?

- Simpler than forward rewriting (no residual)
- Cancellations happen during rewriting
- But expressions may explode

\[ 8z_3^{(1,2)} = 8x_1x_5 \]
\[ 8z_3^{(3,4)} = 8x_1x_2x_3 \]
\[ 8z_3^{(5,6)} = 8a_1b_1a_0b_0 \]

\[ 4z_2^{(1,2)} = 4x_1 + 4x_5 - 8x_1x_5 \]
\[ 4z_2^{(3,4)} = 4x_1 + 4x_2x_3 - 8x_1x_2x_3 \]
\[ 4z_2^{(5,6)} = 4a_1b_1 + 4a_1a_0b_1b_0 - 8a_1a_0b_1b_0 \]

\[ F_{\text{spec}} = 8z_3 + 4z_2 + 2z_1 + z_0 \]
\[ F_{\text{spec}}^{(1,2)} = 4x_1 + 4x_5 + 2z_1 + z_0 \]
\[ F_{\text{spec}}^{(3,4)} = \ldots \]

Still Linear!!
Example: 4-bit CSA-multiplier

- Compare size of intermediate expressions
- Individual bits vs. an entire expression ($\text{Sig}_{\text{out}} \rightarrow \ldots \rightarrow \text{Sig}_{\text{in}}$)
Backward Rewriting - Summary

- No residual expression!
  - But the cut expression can explode (*fat belly* issue)
  - Choice and ordering of cuts during rewriting affects performance

- Issues:
  - Minimize the “fat belly”, the size of largest expression (memory)
  - Handling complex gates
  - Provide cuts in AOI gate
Performance for original & lightly synthesized designs

- Synthesis performed by ABC “resyn”
- Verified designs
  - Multipliers, matrix multiplier, \( A*B+C \), squarer, etc.
  - Up-to 5 million gates
  - 256+ bit-widths
- ~Linear CPU time
- Memory: quadratic in # gates
Extract arithmetic functions with unknown boundaries
- Assume that PO boundary is known but no boundary for PIs
- Backward rewriting

Spectral method
- Examine distribution of weights (coefficients) of polynomial terms during rewriting
- Determine arithmetic function corresponding to a sub-expression
- Based on its coefficients
\( n \)-bit adder

\( i = \) bit position of result
\( C(i) = 2^i, \) coefficient a bit \( i \)
\( N(i) = \# \) terms with coeff \( C(i) \)

\[ A = a_0 + 2a_1 + 4a_2 \]
\[ B = b_0 + 2b_1 + 4b_2 \]
\[ A + B = a_0 + 2a_1 + 4a_2 + b_0 + 2b_1 + 4b_2 \]

\( N(i) = 2, \ i \in (0, 1, 2..., n-1) \)
Arithmetic Spectrum – Linear Functions

- *n*-bit word shifting
  \[ N(i) = 1, \ i \in (0,1,2...,n-2) \]

- **Multiplexer**
  \[ Z_{MUX} = W - 2s \cdot W \]
  \[ N_1(i) = 2, N_2(i) = 1, \ i \in (0,1,2...,n-1) \]
Spectrum – Multiplier (nonlinear)

- **Multiplier**

\[ N_i = \begin{cases} 
  i + 1 & \text{if } i \leq n/2 - 1 \\
  n - i - 1 & \text{if } i \geq n/2 
\end{cases} \]
Abstraction – Spectral Method

Gate-Level netlist

Eqn

last eqn?

yes

exit()

no

compute expression of next cut

split into subexpressions

One variable spectrum

Two vars spectrum

... n vars spectrum

match known spectrum(s) ?

yes

generate word-level structure

no

algebraic rewriting
Does the circuit structure affect the spectrum?

- No, it affects rewriting performance, but not the spectrum

Example: 3-bit Booth multiplier vs. CSA multiplier

- Diagram: single-, double-, triple-variable terms (left to right)

Expression with 1-variable terms

\[ F_{spec} = z_0 + 2z_1 + 4z_2 + 8z_3 + 16z_4 + 32z_5 \]
3-bit Mults

Rewriting progress

20%

40%
Spectrum – Booth and CSA Multiplier

- 3-bit Mults

Rewriting progress

80%

100%

Non-linear word detected!
Spectrum – 3-operand Multiplier

- Spectrum for *arbitrary* arithmetic function?
  - 3-operand multiplier $A*B*C$
  - *Addition, multiplication* or combination
    - Multiply-Accumulator (MAC)

- Word abstraction from expression
  - Pairing signals
  - Need topological analysis

\[ \ldots + 4x_1x_5 + 2x_1x_3 + x_3x_7 + 2x_5x_7 + \ldots \]
\[ = \ldots + (2x_1+x_7)(2x_5+x_3) + \ldots \]
Multiply Accumulator \((F = A \times B + C)\) 

- Can we identify the adder and the multiplier?  
  - Adder or multiplier may not exist after synthesis
- We can tell that there is an *addition* and a *multiplication* 
  - Identify the upper boundary of function \(F\)
- What we cannot do: identify the adder or multiplier 
  - Structural level

**Example : MAC** 

- 2-bit multiplier following 4-bit adder
Arithmetic Spectrum - MAC

MAC

Linear word!

Addition and multiplication detected

EPFL Workshop 2015
8- to 128-bit MAC

Limitations

- Need to know output bits
- Computing spectrum is expensive

<table>
<thead>
<tr>
<th>size k</th>
<th>#. gates</th>
<th>pre-ordering</th>
<th>Addition</th>
<th>Multiplication</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>529</td>
<td>0.01 s</td>
<td>0.01 s</td>
<td>0.22 s</td>
<td>0.24 s</td>
</tr>
<tr>
<td>16</td>
<td>2089</td>
<td>0.01 s</td>
<td>0.03 s</td>
<td>2.71 s</td>
<td>2.75 s</td>
</tr>
<tr>
<td>32</td>
<td>8281</td>
<td>0.03 s</td>
<td>0.11 s</td>
<td>50.7 s</td>
<td>51.00 s</td>
</tr>
<tr>
<td>64</td>
<td>32953</td>
<td>0.07 s</td>
<td>0.47 s</td>
<td>1028.9 s</td>
<td>1029 s</td>
</tr>
<tr>
<td>80</td>
<td>51432</td>
<td>0.12 s</td>
<td>0.76 s</td>
<td>3049.5 s</td>
<td>3050 s</td>
</tr>
<tr>
<td>96</td>
<td>74008</td>
<td>0.15 s</td>
<td>1.27 s</td>
<td>2.0 hrs</td>
<td>2.0 hrs</td>
</tr>
<tr>
<td>112</td>
<td>100681</td>
<td>0.21 s</td>
<td>1.62 s</td>
<td>3.7 hrs</td>
<td>3.7 hrs</td>
</tr>
<tr>
<td>128</td>
<td>131465</td>
<td>0.27 s</td>
<td>2.23 s</td>
<td>6.6 hrs</td>
<td>6.6 hrs</td>
</tr>
</tbody>
</table>

**TABLE 1.**  WORD-ABSTRACTION EVALUATION USING
MULTIPLY-ACCUMULATOR  S = SECONDS, HRS = HOURS
Thank You!