Session 3: Test and Verification

- Shiva – a Hybrid Constraint Solver for Verification and Test: Tim Cheng (15 min.)
- Embedded flash memory testing – CW Wu (15 min.)
- Fault Diagnosis – Shi-Yu Huang (15 min.)
- Verification – Jing-Yang Jou (15 min.)
- Analog/Mixed-Signal Testing - Juin-Lang Huang (15 min.)
- Discussion – Tim Cheng
Shiva – A Hybrid Constraint Solver for Verification and Test

Tim Cheng
Univ. of California
Santa Barbara, CA
Outline

- SHIVA – A hybrid constraint-solver for verification
- A self-referential technique for gate-level arithmetic circuit verification
Efficient constraint solver for Boolean and arithmetic domains

- ATPG for Boolean control
- Presburger Math for Arithmetic Data-path
- Tightly integrated
- Finds best partition dynamically
- Can handle limited non-linear math (not implemented yet)
- Has application in property checking and function vector test generation
- 10-1000x improvement over Boolean ATPG
SHIVA Architecture

Front-End
- Parser
  - ADD
  - Generate Cuts

Back-End
- ATPG Solver
- Solver Manager
- Arithmetic Solver

Constr. Module
- Generator
  - Templates
- Learned Constr.
  - Analyze

Templates

ADD
Generate Cuts
Overall Solver

- **Boolean Space**
  - Pruned by Constraint Propagation

- **Arithmetic Space**
  - Pruned by Constraint Propagation

- Decision
- Conflict
- Boolean / Arithmetic Boundary
Example

- Non-Linear system
  - \[ g = (f == e) \land (e = ((\text{sel'.c} \lor \text{sel.d}) \land (c = a1 + a2) \land (d = b1 + b2))) \]

- Value Assignments on \(\text{sel}\) and \(g\) by ATPG
  - Eg: \{\text{sel}, \text{g}\} = \{0,1\}
  - Solution is IS_Satisfiable \((f == a1 + a2)\)

- Advantage
  - Only decision points here are 1 bit \(g\) and 1 bit \(\text{sel}\)
  - If comparator was in Boolean solver, we have 64 decision points.
Hybrid Solver

- ATPG for search in Boolean Space
  - Sequential ATPG
  - Modified FAN with several enhancements
    - Improved back-trace, Conflict analysis & Dynamic learning
- Global arithmetic solution for Arithmetic space
  - Based on value requirements for given property and requirements generated during ATPG
- Efficient bounding techniques using
  - Implication in Boolean Logic
  - Constraint Propagation in Arithmetic
  - Learned constraints from the Constraint Module
Data-Path Constraint Solving

- **System of equations**
  - Variable are inputs and outputs
  - Equations are derived from
    - data-path functionality & constraints

\[ a + 2b + c = o; \]
\[ a \leq 4; \quad a > 0; \]
\[ b \leq 4; \quad b > 0; \]
\[ c \leq 8; \quad c > 0; \]
\[ o = 2; \quad \text{(constraint)} \]

Solution:
\{a,b,c\} \in \{(2,0,0), (0,1,0), (0,0,2), (1,0,1)\}
Arithmetic Solver

- Based on Omega Library (U. Maryland, Coll. Park.)
  - Based on Presburger Arithmetic
  - Arithmetic based on set algebra
- Can handle mod-2 arithmetic
- Handles \{ \geq, \leq, <, =, >, -, +, \neg \}
- Handles multiply by const only.
- Can handle shift, div, rotate
  - Not implemented yet
  - Non-linear operations can increase complexity
  - Extend solver to non-linear operations in future.
Preliminary Experiments

- GCD circuit
- Used Assignment Decision Diagram (ADD) to extract properties
- 8 properties extracted
- Hybrid approach was compared with ATPG
ADD Generation
An Example

Fpuwakeupsrccp1l || Fpucancmd1[1:1]
qclk10
Fpuwakeupsrccregh
Fpuwakeupsrccp1l[8:64:81]
Fpuwakeupsrccp1l[8:64:81]
qclkfpudivexpd1
w1
Fpuwakeupsrccp1l
L
DFF
Fpuwakeupsrccp1l
Fpuwakeupsrccp1l
Fpuwakeupsrccp1l
Fpudivsignfp2h
Fpudivexpfp2h
w2
w3
w4
DFF
DFF
1
0x1FFE
0x17FFE
L
Fpufrcpainstfp1h
Fpufrcpainstfp1h
## Results on GCD

<table>
<thead>
<tr>
<th>Prop No</th>
<th>Shiva</th>
<th>Boolean ATPG</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4-bit</td>
<td>8-bit</td>
</tr>
<tr>
<td>1</td>
<td>0.03</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0.12</td>
<td>0.01</td>
</tr>
<tr>
<td>3</td>
<td>0.19</td>
<td>0.67</td>
</tr>
<tr>
<td>4</td>
<td>0.2</td>
<td>0.48</td>
</tr>
<tr>
<td>5</td>
<td>0.21</td>
<td>0.59</td>
</tr>
<tr>
<td>6</td>
<td>0.19</td>
<td>0.65</td>
</tr>
<tr>
<td>7(F)</td>
<td>1.19</td>
<td>1.17</td>
</tr>
<tr>
<td>8(F)</td>
<td>0.61</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>2.29</td>
<td>7236.8</td>
</tr>
<tr>
<td>10</td>
<td>1.6</td>
<td>4125.54</td>
</tr>
</tbody>
</table>
On-Going Work/Research Plan

- Improve Sequential ATPG to fastest known SAT/ATPG solver
- Continue the development of the hybrid solver
- Constraint extraction using ADD/EFSM
- Use extracted constraints for efficient property checking
- Show functional TG is feasible using SHIVA
Outline

- SHIVA – A hybrid constraint-solver for verification
- A self-referential technique for gate-level arithmetic circuit verification
Gate-Level Arithmetic Circuit Verification

- Problem: Is the given gate-level netlist implementing the function of $A*B*C+A*$E?
- Key issues:
  - Difficulty of verifying individual arithmetic operators
  - Operand ordering problem
  - Merged arithmetic
  - Circuit transformation by arithmetic relation

Self-referential technique: addresses the first three issues at the gate level.
Varieties of Arithmetic Architectures

Example: Multiplier

Different addition reduction trees with/without recoding.
Self-Referential Verification – Basic Idea

- Use a gate-level implementation to verify itself
- Utilize functional equations of the intended arithmetic function
- Apply inductive definition to divide the problem into a set of sub-problems
- Apply logic equivalence checking to solve each sub-problem
- Example: \( A \times B \)
  \[
  (a_{n-1} a_{n-2} \ldots a_0) \times (b_{n-1} b_{n-2} \ldots b_0)
  = (a_{n-1} a_{n-2} \ldots a_0) \times (0 b_{n-2} \ldots b_0) + (a_{n-1} a_{n-2} \ldots a_0) \times (b_{n-1} 0 \ldots 0)
  \]
Multiplier Verification – Step I

If the reduced multiplier \((a_{n-1} a_{n-2} \ldots a_0) \times (0 b_{n-2} \ldots b_0)\) is correct, then the original multiplier \((a_{n-1} a_{n-2} \ldots a_0) \times (b_{n-1} b_{n-2} \ldots b_0)\) is also correct.
Multiplier Verification – Step II

Equivalence Checking

If the reduced multiplier \((a_{n-1} a_{n-2} \ldots a_0) \times (0 0 b_{n-3} \ldots b_0)\) is correct, then the original multiplier \((a_{n-1} a_{n-2} \ldots a_0) \times (b_{n-1} b_{n-2} \ldots b_0)\) is also correct.
More Examples

- **Triple product** \((x \times y \times z)\).

\[
(0.0x_{n-1-l}..x_0) \times (y_{n-1}..y_0) \times (z_{n-1}..z_0) \\
=(0.0x_{n-2-l}..x_0) \times (y_{n-1}..y_0) \times (z_{n-1}..z_0) + (0.0x_{n-1-l}..0) \times (y_{n-1}..y_0) \times (z_{n-1}..z_0)
\]

- **Inner product** \((x \times y + z \times w)\).

\[
(x_{n-1}..x_0) \times (y_{n-1}..y_0) + (0..0 z_{n-1-l}..z_0) \times (w_{n-1}..w_0) \\
= (x_{n-1}..x_0) \times (y_{n-1}..y_0) + (0..0 z_{n-2-l}..z_0) \times (w_{n-1}..w_0) \\
+ (0..0 z_{n-1-l} 0..0) \times (w_{n-1}..w_0) \\
(0..0x_{n-1-l}..x_0) \times (y_{n-1}..y_0) \\
=(0..0x_{n-2-l}..x_0) \times (y_{n-1}..y_0) + (0..0x_{n-1-l}..x_0) \times (y_{n-1}..y_0)
\]
Which Operand to Decompose?

- Decompose the one with small fanout cone first.
  - Results in simpler equivalence checking problems.
  - Handles the operand ordering problem and produce the correct decomposition properly.
Which Operand to Decompose?

- Analyze the fanout cone of bit variables
- Decompose the ones with small fanout cone first.

An 8-bit multiplier

fanout cone of a's MSB

fanout cone of b's MSB
Experimental Results - A*B

Platform: P3 733Mhz computer with 256MB memory

- addstep
- csatree
- cla
- booth
- wallace

Time (sec) vs Multiplier size (bit)
Memory Usage - A*B

*Verification of C6288 takes only 4.67sec and 0.94MB!
Experimental Results – A*B+C

Memory usage is between 0.5M to 4.6M
Experimental Results

*Memory usage is between 0.4M to 2.0M*
Self-Referential Verification

- Heuristics proposed to simplify the resulting equivalence checking problems.
- Capable of verifying a large class of arithmetic circuits.
- Publications: ICCAD2001, DAC2002