Formal Equivalence Checking

Logic Verification

- Full-chip functional verification to verify the correctness of the design
- Equivalence checking to verify that actual implement = design
- Equivalence checking to make sure test model is appropriate
- Timing and power are analyzed at layout level
Boolean Equivalence Checking

- Check for "structural equivalence"
- Require one-to-one latch correspondence
  » Partition design into independent combinational cone
- Verify each combinational logic cone independently
- Impose additional restrictions on custom design

Approaches

- Step 1 - Extract gate-level model from transistor-level schematics
  - Explicit - pattern matching + symbolic analysis
  - Implicit - symbolic simulation/analysis
- Step 2 - Check equivalence between two (combinational) logic cones
  - OBDD-based approach
  - SAT-based approach
Model Extraction

- **Model extraction**
  - Symbolic boolean analysis + pattern matching
  - Need to recognize precharge, self-timed circuitry, latches, sometimes even memory cells
  - Need to find an “acceptable” gate-level model
    - May not be exactly as what the original design intended
  - Useful for testing as well

Symbolic Analysis

- **For each CCC component**
  - consider all lines to the transistor gates as inputs
  - consider all lines to the transistor gates in other components as outputs

- **Steady state response**
  - given a set of input values, compute steady output values

- **Each component behaves like an independent function block**
  - When extracting, we handle each component separately
Symbolic Analysis of Transistor Network

- Inputs and node states can be “symbols”
  - a symbol can be either 0 or 1
  - if all inputs are symbols, we try to compute the boolean function given by the network
    \[ f = ab + cd, \text{ etc.} \]
- Basic idea:
  - Develop the symbolic equation for every CCC
  - Synthesize the equation into gate netlist
  - Connect the netlists together to form the final circuit

Construct Gate Level Model

- Symbolic Boolean Analysis gives us for CCC
  - Each output node as a boolean function of inputs
  - Try to convert that function into Gates
- Gate primitives used
  - AND, OR, NOT, MERGE, ENABLE
  - Operate on a 4-value system
    \[ 0, 1, Z, X \]
- Apply *heuristics* to prune the gate level model
Construction Gate Level Model 2

• 0, 1, Z, X
  ➢ Z means “no value”
  ➢ X means “conflicting values”

• ENABLE
  ➢ Acts as an optimistic TSD
  ➢ Can use TSD instead
    ✓ The result will be pessimistic

• MERGE
  ➢ Acts as LUB

Pattern Matching + Path Enumeration

• Ex. GateMaker (IBM, Motorola)
  ➢ Preserve schematic structure better

• Classify circuit into 4 families
  ➢ Combinational Static
  ➢ Differential Cascode Voltage Switch
  ➢ Pass Gate Logic
  ➢ Dynamic Domino

• Recognize special design styles
  ➢ Keeper
  ➢ Feedback logic
Slide # 11

Patterns

Slide # 12

Suppose Extraction Works

- EQ checker works on 2 gate level models
  - Identify latch correspondences
  - Incrementally prove permissible signal pairs
  - Use ATPG, BDD, or both
Identify Latch Correspondences

- Manually or automatically?
- Manually – much overhead to the user
- Automatically – various heuristics are applied
  - Identify a latch point
  - Identify structural similarity
  - Utilizes design information (hierarchy, names, etc.)

Checking EQ of 2 Logic Cones

- ATPG-based approach
  - To prove that the s-a-0 is a redundant fault
  - Less memory constraint and hence, scale better
- (SAT-based approach)
- OBDD-based approach
  - More memory constraint
Structural Similarity Helps

- Structural similarity allows EQ to proceed incrementally
  - Divide and conquer
  - Employ various heuristic to guess matching points
  - Most EQ tools explore structural similarity to speed up the process

Local BDD can help

- When proving $a = a'$, the cones of logic may be very different
  - ATPG approach becomes inefficient
  - Similarity can’t be found
- Use local BDDs - BDDs based on internal points
  - Local BDDs can quickly show the equivalence of $a$ and $a'$ given that all supporting points are equivalent
  - Proceed backward with a limited # of levels to search for local-BDD-based equivalent
BEQ IS Practical!!

- BEQ is practical
  - Has been applied to full-chip μ-processors
  - Commercially available

- BEQ check for “strong equivalence;” however
  - Consistency may be enough
    - Both behave the same in the care functional space
    - Equivalence up to limited clock cycles may be enough

- BEQ will have its difficulty
  - on memory designs
  - when there is no structural similarity (on large and complex cones)
  - when there is no 1-to-1 latch correspondence
Arrays

- Custom design at transistor level
- Large in Size (Millions of Transistors)
- High-Performance Sequential Logic
- Examples: Cache, Tag, BAT, TLB, Regs, Status Array, etc.

Symbolic Simulation

- Traditional 0/1 vector simulation
  - Input 0/1 vector; Results in 0/1
- Symbolic Simulation
  - Input arbitrary symbols; Results in logic functions (often represented as OBDDs)
  - Bounded by a constant number of clock cycles
Why It Is Good For Memories?

- Symbolic simulation of an address $A = [a_1, a_2, ..., a_n]$ with data $D = [d_1, d_2, ..., d_m]$ can cover all memory cells at once
  - In contrast, imagine you perform 0/1 simulation on a large memory block
  - Today, an array easily has millions of cells
- Memories are often custom designs that contain sequential behavior
  - Contain dynamic logic
  - Hard to model at gate-level
  - LEQ does not apply

Symbolic Trajectory Evaluation

- X’s at inputs are “don’t care”
- X’s during simulation are “don’t know”
- Allow more flexibility to partition functional space
  - Easy handling of tri-state buffers, bus, etc.
  - Improve symbolic simulation efficiency (simulate only the case functional space)
  - Bounded by certain clock cycles
Trajectory Evaluation

- Based on three-value logic: \{0, 1, X\}

State of this inverter is a 2-tuple \(\langle A, B \rangle\) where both A, B can have a value in \{0, 1, X\}.

<table>
<thead>
<tr>
<th>t=0</th>
<th>t=1</th>
<th>t=2</th>
<th>t=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle A, B \rangle)</td>
<td>(\langle 0, X \rangle)</td>
<td>(\langle X, 0 \rangle)</td>
<td>(\langle 0, X \rangle)</td>
</tr>
</tbody>
</table>

**A valid trajectory**

<table>
<thead>
<tr>
<th>t=0</th>
<th>t=1</th>
<th>t=2</th>
<th>t=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle A, B \rangle)</td>
<td>(\langle 0, X \rangle)</td>
<td>(\langle X, 0 \rangle)</td>
<td>(\langle 0, 0 \rangle)</td>
</tr>
</tbody>
</table>

**A valid trajectory**

<table>
<thead>
<tr>
<th>t=0</th>
<th>t=1</th>
<th>t=2</th>
<th>t=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle A, B \rangle)</td>
<td>(\langle 0, X \rangle)</td>
<td>(\langle X, 1 \rangle)</td>
<td>(\langle 0, X \rangle)</td>
</tr>
</tbody>
</table>

**An invalid trajectory**

This cannot be!

Assert: If A=1 then B=0

---

Correct Trajectory

- Assert: If (A is 0 from 0 to 1) then (B=1 from 1 to 2)

\[1\text{ unit delay}\]

<table>
<thead>
<tr>
<th>t=0</th>
<th>t=1</th>
<th>t=2</th>
<th>t=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle A, B \rangle)</td>
<td>(\langle 0, X \rangle)</td>
<td>(\langle X, X \rangle)</td>
<td>(\langle X, X \rangle)</td>
</tr>
</tbody>
</table>

**Weakest sequence satisfying Antecedent**

<table>
<thead>
<tr>
<th>t=0</th>
<th>t=1</th>
<th>t=2</th>
<th>t=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle A, B \rangle)</td>
<td>(\langle 0, X \rangle)</td>
<td>(\langle X, 1 \rangle)</td>
<td>(\langle X, X \rangle)</td>
</tr>
</tbody>
</table>

**Weakest satisfying trajectory**

<table>
<thead>
<tr>
<th>t=0</th>
<th>t=1</th>
<th>t=2</th>
<th>t=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle A, B \rangle)</td>
<td>(\langle X, X \rangle)</td>
<td>(\langle X, 1 \rangle)</td>
<td>(\langle X, X \rangle)</td>
</tr>
</tbody>
</table>

**Weakest sequence satisfying Consequent**

Note that: \(1 \subseteq X\), and \(0 \subseteq X\)
Incorrect Trajectory

Assert: If (A is 0 from 0 to 1) then (B=1 from 1 to 2)

Weakest sequent satisfying Antecedent

Weakest sequent satisfying Consequent

STE Assertion

• Express each array operation as “Antecedent => Consequent”
  • where Ant and Con are written in a functional language following a restricted linear time logic
  • The number of assertions for a design may vary depending on the person who creates them
A Write Assertion Example

- At time T,
  - If \((A1 = A2)\) then \(Array[A1] = D2\)
  - else \(\{ Array[A1] = D1 \text{ and } Array[A2] = D2 \}\)
- Describe how outputs behave based on symbolic controls during CLK high
- Describe output behavior during CLK low

STE Methodology

- Assertions capture the high level functionality
- RTL can be verified independently
  - achieve both logic and functional verification w.r.t. the set of assertions
STE = EQ

- Extract assertions from RTL
- Do not verify RTL independently
- Provide a bounded-cycle sequential equivalence checking methodology

An Array Bit Cell

Write enable from L1 latch
Read enable from L2 latch
dout latched in a L1 latch
din and din_b from L1 latch

4 unit delays
9 unit delays
Slide # 31

Bit Cell Timing Diagram

Slide # 32

A Verification Example

RTL model

module(C1,C2,Ain,Bin,Cout)
input Ain,Bin,C1,C2;
output Cout;
wire and_out;
reg A.L1, A.L2, B.L1,B.L2, C.L1, C.L2
assign and_out = A.L2 & B.L2;
always @(C1 or Ain or Bin or and_out)
if (C1)
   A.L1 = Ain; B.L1 = Bin; C.L1 = and_out;
always @(C2 or A.L1 or B.L1 or C.L1)
if (C2)
   A.L2 = A.L1; B.L2 = B.L1; C.L2 = C.L1;
assign Cout = C.L2;
endmodule
Verifying The Schematic

Derive

\[ C1 \land (A \land B) \lor \neg C1 \land C \]

from the RTL automatically

Overall Verification Strategy

Carefully timed read and write control logic and data conditioning
**Results - Motorola Versys2**

<table>
<thead>
<tr>
<th>Array block</th>
<th>Bit cells</th>
<th>Latches</th>
<th>Control logic transistors</th>
<th>Assertions runtime (hrs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>73,728</td>
<td>1,348</td>
<td>69,000</td>
<td>11–12</td>
</tr>
<tr>
<td>B</td>
<td>24,576</td>
<td>1,612</td>
<td>87,500</td>
<td>15–16</td>
</tr>
<tr>
<td>C</td>
<td>3,968</td>
<td>357</td>
<td>31,000</td>
<td>15–16</td>
</tr>
<tr>
<td>D</td>
<td>24,576</td>
<td>935</td>
<td>44,000</td>
<td>16–17</td>
</tr>
<tr>
<td>E</td>
<td>131,072</td>
<td>330</td>
<td>177,500</td>
<td>19–20</td>
</tr>
<tr>
<td>F</td>
<td>88,704</td>
<td>445</td>
<td>39,500</td>
<td>12–13</td>
</tr>
<tr>
<td>G</td>
<td>50,688</td>
<td>798</td>
<td>158,000</td>
<td>11–12</td>
</tr>
<tr>
<td>H</td>
<td>71,680</td>
<td>276</td>
<td>60,000</td>
<td>15–16</td>
</tr>
<tr>
<td>I</td>
<td>21,824</td>
<td>1,152</td>
<td>27,000</td>
<td>2–3</td>
</tr>
<tr>
<td>J</td>
<td>1,024</td>
<td>0</td>
<td>8,950</td>
<td>1.5–2</td>
</tr>
<tr>
<td>K</td>
<td>4,096</td>
<td>0</td>
<td>7,900</td>
<td>5–6</td>
</tr>
<tr>
<td>L</td>
<td>8,512</td>
<td>0</td>
<td>8,800</td>
<td>5–6</td>
</tr>
<tr>
<td>M</td>
<td>256</td>
<td>0</td>
<td>4,590</td>
<td>6–7</td>
</tr>
<tr>
<td>N</td>
<td>512</td>
<td>0</td>
<td>1,250</td>
<td>1–2</td>
</tr>
<tr>
<td>O</td>
<td>2,096</td>
<td>0</td>
<td>32,400</td>
<td>1–2</td>
</tr>
<tr>
<td>P</td>
<td>4,192</td>
<td>0</td>
<td>6,050</td>
<td>4–5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Array block</th>
<th>Validation time (person-months)</th>
<th>Discrepancies</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>H</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>I</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>J</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>K</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>L</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>M</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>N</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>O</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>P</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

**In Summary**

- **Symbolic simulation is efficient for embedded array verification**
  - Symbolic addressing covers all space at once
  - Can be used for both logic and functional verification (for array blocks)
  - Has been applied to very large array designs in practice
- **The methodology is not as rigorous as LEQ**
  - LEQ checks for strong equivalence
  - Symbolic simulation checks for consistency based on a pre-defined range of clock cycles