Assertion Synthesis
Enabling Assertion-Based Verification For Simulation, Formal and Emulation Flows
Assertion-Based Verification adopted by leading design companies

- Most effective approach to reduce chip failure
  - Corner case bugs
  - Identify testing coverage holes
- Reduce overall verification effort

Not enough assertions and coverage goals written

Not getting the benefit of ABV methodology
- Functional errors still #1 chip failure cause and cost
- Need an effective approach to create assertions
Blackbox Assertions

- Capture End to End behaviors using interface signals
  - Off the shelf VIP for standard bus protocols
  - An alternative to blackbox checkers

- Issues
  - Difficult to create and maintain due to complexity
    - Duplicates RTL logic in assertion language
    - More temporal because of limited use of signals
  - Slow down simulation (due to multi-cycle temporal properties)
  - Large area overhead in Emulation/FPGA prototype
  - Difficult to debug failed assertions

- DV write them based on functional spec
White-box Assertions

- **Capture RTL behaviors using internal signals**
  - Low overhead in simulation runtime and emulation area
  - Reduce debug turn-around by pinpoint error source
  - Easier for formal tools to converge
  - Identify implementation specific assertions and coverage

- **Issues**
  - Require designer time and expertise
  - Need orthogonal perspective
    - Which combination of signals to use?
  - Beyond functional specification
    - Assertions and coverage not conceived/missed by human

- **Not enough whitebox assertions from designers**
Example Properties

Useless property:
req |→ ##1 busy
(!req & rdy) |→ ##1 !busy

Orthogonal White-box property:
rdy |→ busy

When rdy is asserted, in the same cycle, bus is always busy. That is, rdy must follows a req.

Another orthogonal white-box property:
req |→ !busy

When req is asserted, in the same cycle, bus is always non-busy. That is, new req is only asserted when rdy for previous req is asserted before.

Blackbox property:
$\text{rose}(\text{req}) |→ !\text{req} \text{ throughout } ##[1:$] \text{ rdy}$;
BugScope Assertion Synthesis Flow

Generate Stimulus to Patch Coverage Holes

Stimulus

RTL Design

BugScope Assertion Synthesis

RTL Simulator

BugScope Properties Report

Review

Functional Coverage

Assertions

RTL Simulator, Formal, or Emulator

Bind Assertions to Uncover Bugs

No Changes in Methodology Required
1. Simulate using NextOp PLI
   - Input: Testbench + Test Vectors + RTL
   - Output: a Nextop database .ndb file for each simulation run

2. NextOp post processing
   - Input: Nextop databases + RTL
   - Output: Nextop properties
     - Properties guaranteed to hold for the given tests
     - A property is an assertion OR its negation is a coverage hole

3. Review/classify properties into assertions and coverages
   - SVA, PSL, Verilog and/or Synthesizable Verilog
Key Features to Make Assertion Synthesis Useful

- Automated
  - discover properties not recognized by engineers

- High Quality of Results
  - 1%-10% of RTL lines
  - Don’t duplicate RTL or other properties

- Efficient Assertions
  - Low runtime overhead for simulation
  - Low area overhead for emulation

- Capacity
  - Full chip assertion synthesis solution

- Seamless integration within existing flows
Primary Use Models

- **Coverage-driven simulation flow**
  - Mainstream methodology

- **Simulation + formal property checking flow**
  - Advance corner-case bug-hunting methodology

- **Emulation/Acceleration Flow**
  - Enable trigger point to reduce debug turn-around

- **Functional Review**
  - Property classification using debugger drives functional review
Coverage-Driven Simulation Flow

Find design errors

Coverage hole/Bug

Simulate
NextOp PLI

Database

NextOp Analysis

Review

Prevent usage errors

Bug

Simulate/
FPGA/
Emulate

Assertions

IP Users/Firmware/Software

Coverage holes detect TB issues
Assertions catch RTL errors

IP Design/Verification

Ship RTL + Assertions as IPs to catch usage errors.
Assertions & Coverage Over Verification Cycle [NVIDIA, DAC’10]

Early Middle Late Verification Cycle

Number of Properties

Number of Simulation Runs

#coverage #assertion
The packet processor includes ~30K lines of Verilog
  The key filtering block includes 3K lines

Total 2000 direct+random tests reaches 100% statement coverage and >95% conditional coverage
Using all **2000** tests, BugScope generates

- **145** properties, average **5%** of RTL lines
- **43** cover properties and **102** assertions

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!(buffer_empty &amp; &amp; filter_fifo_rd)</td>
<td>No FIFO underflow</td>
</tr>
<tr>
<td>{valid, sop}!=0</td>
<td>-&gt;@pkt_length!= pkt_length</td>
</tr>
<tr>
<td>onehot0( {key_pkt, bypass_pkt, invalid_tag} )</td>
<td>mutual exclusive type</td>
</tr>
<tr>
<td>!(eop &amp; &amp; state == DATA1)</td>
<td>A special size runt pkt is not tested</td>
</tr>
<tr>
<td>!(multicast_pkt &amp; &amp; cur_multicast_pkt)</td>
<td>Need to test back-to-back multicast pkts</td>
</tr>
</tbody>
</table>
Properties on Filtering Block [DVCon’11]

- Using all 2000 tests, BugScope generates
  - 145 properties, average 5% of RTL lines
  - 43 cover properties and 102 assertions

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert ! (buffer_empty &amp;&amp; filter_fifo_rd)</td>
<td>No FIFO underflow</td>
</tr>
<tr>
<td>assert {valid, sop}!=0</td>
<td>-&gt;@pkt_length!= pkt_length</td>
</tr>
<tr>
<td>assert onehot0( {key_pkt, bypass_pkt, invalid_tag} )</td>
<td>mutual exclusive type</td>
</tr>
<tr>
<td>cover (eop &amp;&amp; state == DATA1)</td>
<td>A special size runt pkt is not tested</td>
</tr>
<tr>
<td>cover (multicast_pkt &amp;&amp; cur_multicast_pkt)</td>
<td>Need to test back-to-back multicast pkts</td>
</tr>
</tbody>
</table>
Simulation + Formal Property Checking

Create Properties

Find corner case issues

Simulate NextOp PLI

Database

NextOp Analysis

Properties

Formal Property Checking

Proof

Bug

Cover patch

Simulation Regression

Formal/Semi-Formal

Assertions and coverage goals created based on simulation

Whitebox properties are more tractable for formal algorithms
Example Customer Scenario

- A complex SOC with >10K tests
  - A semi-formal tool is used to drive stimulus
  - Designer wrote ~5 assertions, no bug found by formal engine
  - SOC is taped-out and used in field for a few years

- Properties based on direct and random simulation
  - RTL ~1.8K lines
  - BugScope reports 117 properties, including 2 manual assertions
    - <2hrs and ~300MB memory

- Properties used as targets for semi-formal tool
  - 1 BugScope assertion is violated
    - pkt_is_good |→ @state == ACCEPT
  - Bug happens when timer wraps around from 32'hffffffff to 32'h0
Bug was harder to find by other means

- Difficult for simulation because it takes 4 billion cycles to wrap around
- End-to-end properties written by engineers cannot be proven
- Extremely difficult to write constraints for formal at module level
- The bug degrades system performance
  - Performance bug is hard to detect with black-box checkers

Solution with BugScope Assertion Synthesis

- Apply formal with large number of white-box properties
- Apply formal at a higher level that includes > 20 modules
  - White-box assertions allow formal to converge
  - Easier to develop constraints at a higher level
Emulation/Acceleration Flow

Create Assertions and Coverage

- Simulate NextOp PLI
- Database
- NextOp Analysis
- Review

Detect Corner Case Bug

- Assertions
- FPGA/Emulate

Simulation based Verification

WhiteBox Assertion as trigger condition or assertions to reduce debug turn-around time
Example Customer Scenario

- Assertions synthesis debug Post-Silicon problem
  - A Complex Networking SoC hangs
  - Traffic pattern too complex to be reproduced in simulation
  - Emulation reproduces the hang but cannot be debugged

- Create properties based on random simulation
  - RTL ~2K lines with 3000+ simulation tests
  - BugScope reports 300 properties

- Add assertions as trigger points in emulation
  - A NextOp assertion triggered
  - clear_cnt <= 19

- Assertion pinpoints the RTL bug
Functional Review Using Properties & Debugger

- Directly find bugs
- Review becomes more objective between design, architect and verification teams
always @(posedge clk.) begin
    if(load_cnt)
        cnt_incr <= 1'b1;
    else if(cnt == 71)
        cnt_incr <= 1'b0;
    if(cnt_incr) cnt <= cnt+1;
    else cnt <= 0;
end

BugScope property:
\[ cnt == 'd72 |-> @cnt == 'd0; \]
Whenever cnt is 72, in the next cycle, cnt is never 73.

Two bugs founds:
1. The priority to load_cnt was incorrect.
2. The code to latch load_cnt when cnt == 71 was missing to handle the scenario correctly.
The packet processing state machine included 4 states:
IDLE-> RUN-> WAIT-> PAYLOAD.

The signal “start” comes from another block, initiates the transaction, and could only be asserted at IDLE state according to the design assumption.

BugScope property:
\[ \text{start } |\rightarrow \text{cur_state} \neq \text{RUN} \]
Whenever start is asserted, cur state is never run. Designer was expecting a property, start |\rightarrow \text{cur_state} == \text{IDLE}

When waveform and code was traced using debugger, the team find out that in certain scenario, the next packet is initiated to transmit before previous packet is completely sent. This is indicated by “start” signal getting asserted in states other than IDLE.

Bug:
This may result into dropping the transactions and un-necessarily retried.
Summary – Assertion Synthesis

- Reduce verification resources
  - productivity gain: Hours vs minutes in creating assertions

- Reduce verification schedule
  - Synthesized assertions to find bugs early, reduce debug time

- Find critical bugs
  - including post-silicon, post RTL freeze, IP integration …

- Verification signoff
  - [http://deepchip.com/items/0485-03.html](http://deepchip.com/items/0485-03.html)
  - [http://deepchip.com/items/0487-06.html](http://deepchip.com/items/0487-06.html)