The Process and Proof for Formal Sign-Off – A Live Case Study

Ipshita Tripathi, Ankit Saxena, Anant Verma, Prashant Aggarwal
Oski Technology, Inc.
Agenda

• Introduction – Formal Sign-off
• A Live Case Study – “Break the Testbench” challenge at DAC 2015
Introduction – Formal Sign-off
End-to-End Formal Enables Formal Sign-off

- Catch corner case bugs early
- Increase verification efficiency
- Replace block-level simulation
- Enable formal sign-off

Assertion-based Verification (ABV)
Formal Apps
Automatic Formal

Complexity & Benefits
Adoption
What is End-to-End Formal?

• Local Assertions: Easier to verify
  – Internal RTL assertions, embedded in RTL

• Interface Assertions: Harder to verify
  – Relate to inputs/outputs
  – E.g. ACE, AXI4, OCP, DDR2, ...

• End-to-End Checkers: Hardest to verify
  – Model end-to-end functionality
  – Often require Abstraction Models to manage complexity
  – Can replace simulation
"Control", "Data Transport" designer size blocks:

- Arbiters of many kinds
- Interrupt controller
- Power management unit
- Credit manager block
- Tag generator
- Scheduler

- Bus bridge
- Memory controller
- DMA controller
- Host bus interface
- Standard interface (PCIe, USB)
- Clock disable unit
Many SoC Blocks Can Be Verified with End-to-End Formal

- Planning at the micro-architectural design stage is critical
- **End-to-End** formal can fully replace simulation for many blocks
Oski Formal Sign-off Methodology

Quality of Formal Depends on all 4 Cs!

End-to-End Checkers

Constraints

Design Under Test (DUT)

Coverage (Code and Functional)

Complexity (Abstraction Models)
End-to-End Formal is Complete

For End-to-End formal to be complete, ideal metrics answer:

• Constraints: Have I unintentionally over-constrained any inputs?

• Complexity: Have all my checkers reached the Required Proof Depth?

• Checkers: Does my list of checkers fully embody the specified behaviour of the design?

We will use Formal Coverage to quantify each of these three questions
Constraints: Ensuring No Unintentional Over-constraints

• Review the list of constraints with the designer

• Validate absence of unintentional over-constraints:
  1. Instantiate of constraints as assertions in simulation
  2. Use cross-proof with neighboring blocks
  3. Use of formal coverage
Complexity: Reaching the Required Proof Depth (RPD)

- Use the 6-step methodology to derive the RPD*

- Use formal coverage to quantify RPD

- All End-to-End checkers need to reach the RPD

Complexity: Using Creative Techniques to Reduce RPD

Resolving complexity challenges using Abstraction Models, or other techniques

- Sequence, counter, reset, floating pulse ...
- Localization, datapath, memory

Interesting Corner-cases

Short-cuts due to Abstraction Models
Checkers: Ensuring Completeness of End-to-End Checkers

• Review the list of checkers with the designer
  – Ensure each output has an End-to-End checker, unless the designer determines the output does not need one
    • For example no need to verify profiling signals or test signals that are not related to design functionality
• Use negative testing (design mutation)
  – Randomly/Intelligently insert design bugs manually to make sure they are caught by existing checkers
  – Verify that every bug found by simulation is also found by existing checkers
• Use formal coverage (proof core) to ensure 100% code coverage
Live Case Study – “Break the Testbench” Challenge at DAC 2015
Multicast Crossbar Design Specifications

- A client can send request along with data to any target

- A client request can go to multiple targets (multicast)

- Each target has an arbiter that determines which client’s request gets forwarded
Design Stats

### Design

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>40</td>
</tr>
<tr>
<td>Outputs</td>
<td>32</td>
</tr>
<tr>
<td>Flops</td>
<td>312</td>
</tr>
<tr>
<td>Lines of RTL code</td>
<td>1,229</td>
</tr>
<tr>
<td>Files</td>
<td>6</td>
</tr>
</tbody>
</table>
English List of End-to-End Checkers

• Arbitration checker on req_out and client_id
  1. Among multiple requests, a request with highest priority type (strict, high, normal) should be seen at output
  2. Among multiple requests of the same priority, a given client should not get a grant twice before the other client has been given a grant

• Consistency checker on req_out and client_id
  – Any output from a target should have had an associated client request (i.e. no spurious outputs are seen)

• Consistency checker on grant
  – Any grant at a client should have had an associated client request (i.e. no spurious grants are seen)

• **Consistency checker on req_data_out**
  – Data seen at target output should be consistent with data seen at client input (i.e. data should not be corrupted, duplicated, reordered or dropped)

• Forward progress checker on req_out
  – A client request should be seen at some target output within finite time

• Forward progress checker on grant
  – A client request should get a grant within finite time
English List of Design Constraints

• If strict or high priority is asserted by a client, request must be asserted by the client

• Once request is asserted by a client; request, strict priority, high priority and data must be held stable until grant is asserted

• Only one client will send a strict priority request at a time
Using Symbolic Variables in Formal Testbench

• 8 clients and 8 targets, each client can send a request to multiple targets
  – Large number of combinations need to be tracked to fully verify the design
• Select symbolic random client and symbolic random target
  – Track all requests from symbolic client to symbolic target
  – All possible combinations exercised by formal tool in single run

```plaintext
##1(sym_target == $past(sym_target))
##1(sym_client == $past(sym_client))
```

Select symbolic random client

Select symbolic random target

MULTICAST XBAR
Data Consistency Checking Using FIFO-Based Scheme

How FIFO based scheme works?

- **Data Drop**
  - DUT
  - FIFO
  - Data Drop

- **Data Duplication**
  - DUT
  - FIFO
  - Data Duplication

- **Data Reorder**
  - DUT
  - FIFO
  - Data Reorder

- **Data Corruption**
  - DUT
  - FIFO
  - Data Corruption

Random data
Data Consistency Checking Using Wolper Coloring Technique

Wolper Coloring Technique Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0^*110^\omega$</td>
<td>If first 1 is seen, next input/output should be 1</td>
</tr>
<tr>
<td></td>
<td>If two 1’s have been seen, only 0’s should be seen</td>
</tr>
</tbody>
</table>

How Wolper coloring technique ($0^*110^\omega$) works?

- **Data Drop – Rule 1 violated**
- **Data Duplication – Rule 2 violated**
- **Data Reorder – Rule 1 violated**
- **Data Corruption – Rule 2 violated**
Constraints: Ensuring No Unintentional Over-constraints

• Reviewed list of constraints with the designer
• Validated using formal coverage (no over-constraints)
  – Line Coverage
    • Total lines: 288
    • Covered lines: 288 (100%)
  – Condition Coverage
    • Total lines: 88
    • Covered lines: 88 (100%)
Complexity: Reaching the Required Proof Depth (RPD)

- Deepest cover point is 5 cycles deep
- Required Proof Depth is 13 cycles
  - Found using 6-step process
  - Not optimistic!

<table>
<thead>
<tr>
<th>Checker</th>
<th>Bound Achieved (in 10min run-time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arbitration checker on req_out and client_id</td>
<td>95 cycles</td>
</tr>
<tr>
<td>Consistency checker on req_out and client_id</td>
<td>Proven</td>
</tr>
<tr>
<td>Forward progress checker on req_out</td>
<td>15 cycles</td>
</tr>
<tr>
<td>Forward progress checker on grant</td>
<td>17 cycles</td>
</tr>
<tr>
<td>Consistency checker on req_data_out</td>
<td>15 cycles</td>
</tr>
</tbody>
</table>

No need to use complexity reduction techniques as Bound Achieved > RPD
Checkers: Ensuring Completeness of End-to-End Checkers

• List of checkers reviewed with the designer

• 73 artificial functional bugs were manually inserted in the DAC 2015 Challenge
  – Checkers found all of them!
---

## Example Bugs Inserted During the Challenge

<table>
<thead>
<tr>
<th>Bug Category</th>
<th>Original RTL</th>
<th>Buggy RTL</th>
<th>#Failing checkers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arbitration Scheme</strong></td>
<td>// arbiter.sv 60 assign high_prio_req = (</td>
<td>str_prio_req) ? {NUM_CLIENT{1'b0}} : (high_prio &amp; req);</td>
<td>// arbiter.sv 60 assign high_prio_req = (</td>
</tr>
<tr>
<td><strong>Arbitration Scheme</strong></td>
<td>// rr_scheme.sv 56 assign shft_req = {req, req};</td>
<td>// rr_scheme.sv 56 assign shft_req = {8'd0, req};</td>
<td>2</td>
</tr>
<tr>
<td><strong>Arbitration Scheme</strong></td>
<td>// rr_scheme.sv 65 for(i = 0; i &lt; (2 * NUM_CLIENT); i = i + 1) begin</td>
<td>// rr_scheme.sv 66 for(i = 0; i &lt; (2 * NUM_CLIENT - 1); i = i + 1) begin</td>
<td>2</td>
</tr>
<tr>
<td><strong>Connectivity</strong></td>
<td>// xbar_8x8.sv 274 .high_prio(high_prio_4),</td>
<td>// xbar_8x8.sv 274 .high_prio(high_prio_3),</td>
<td>4</td>
</tr>
<tr>
<td><strong>Grant generation</strong></td>
<td>// one_dly.sv 71 assign gnt_i = has_data ? outgoing_data : 1'b1;</td>
<td>// one_dly.sv 71 assign gnt_i = has_data ? outgoing_data : 1'b0;</td>
<td>2</td>
</tr>
<tr>
<td><strong>Wrong operator</strong></td>
<td>// target.sv 89 assign t2c_grant = {NUM_CLIENT{ext_grant_pp}} &amp; arb_gnt_pp;</td>
<td>// target.sv 89 assign t2c_grant = {NUM_CLIENT{ext_grant_pp}} &amp; &amp; arb_gnt_pp;</td>
<td>8</td>
</tr>
</tbody>
</table>
Non-bug #1: Round Robin Arbiter

Initial Value Change

Original RTL:
for(i = 0; i < (2 * NUM_CLIENT); i = i + 1)

Modified RTL:
for(i = 1; i < (2 * NUM_CLIENT); i = i + 1)

\(i=1\) makes an RTL optimization to use an n-iteration loop instead of an n+1-iteration loop

- Makes the design slightly better, area-wise
- No functional impact
Non-bug #2: Grant is 0 After Counter Reaches Threshold

Original RTL:
assign high_prio_gnt = tmp_high_prio_gnt;

Modified RTL:
assign high_prio_gnt[0] = tmp_high_prio_gnt[0] &
  ~(my_bug_delay == 1024'b123456789101213);

always @ (posedge clk) begin
  if (rst) my_bug_delay <= 1024'b0;
  else my_bug_delay <= my_bug_delay + 1'b1;
end

• Bug was inserted with a “malicious intent”
  – Used knowledge of the verification methodology to specifically change the design, such that the defect cannot be caught by the test-bench
  – Defeats the purpose of “true” verification i.e. find all "naturally occurring" bugs

• Increased the Required Proof Depth, making bug impossible for formal (and sometimes simulation) to find, without using Abstraction Models
Summary

• Significant design blocks in SoC, processor and networking chips can be verified with formal

• Formal sign-off offers ultimate confidence in verification - No bug left behind

• Formal sign-off can be achieved by ensuring
  1. No unintentional over-constraints
  2. List of checkers is complete
  3. All checkers reach Required Proof Depth