**Systematic Speedup Techniques for Functional CDC Verification Closure**

Sulabh Kumar Khare, Ashish Hari

Mentor Graphics Corporation

### What is Functional CDC Verification
- Every Clock Domain Crossing (CDC) synchronizer must follow some protocol for functional correctness and to avoid data loss.
- Verification of synchronizer protocols using simulation/ formal techniques is called functional CDC verification.

### Why Functional CDC Verification
- Modern SOCs depend on multiple asynchronous clock domains to meet increasing high performance and low power needs.
- Adding a synchronizer alone does not guarantee that the CDC signal will pass through reliably, without getting lost.
- Example of protocol violation (data loss) in a 2-DFF synchronizer:

![2-DFF Synchronizer Diagram](image)

### Existing Functional CDC Verification Flow
- Starts after the completion of static CDC analysis:
  1. CDC Analysis
  2. Generate Assertions
  3. Simulation
  4. Formal Verification
  5. Formal Debug

- Generates properties/assertions for CDC paths, which are then verified using simulation or formal techniques.
- Results are separately debugged in a Formal or a Simulation environment.

### Issues With The Existing Flow
- Noise in results due to redundant checking
  - Example of synchronizer with an enable condition for data transfer:

![Noise Example Diagram](image)

- Waveform for protocol check in the absence of an enable condition:

![Waveform Diagram](image)

- Inaccurate verification with variation in clock frequency:

![Inaccurate Verification Diagram](image)

- Increased time to closure due to missing functional coverage
- Limited debug capabilities
- Inconsistent setup for formal verification
- Inaccurate results due to single clock assertion:

```
property data_stable(data, rx_clock, reset, areset, NUM_CYCLES);
  @posedge rx_clock disable iff(areset)
  !reset && $changed(data) |=> $stable(data)[*(NUM_CYCLES-1)];
endproperty
```

### Proposed Functional CDC Verification Flow
- Starts with the CDC analysis step and applies techniques to accelerate the verification closure:
  1. Formal Debug
  2. Simulation Debug
  3. Formal Verification

- Brings functional CDC verification debug closer to CDC analysis

### Speedup Techniques
- Check assertions only when required
  - Use design analysis to infer the enable condition for data transfer
  - Waveform for protocol check with the enable condition s3&6:

![Waveform Diagram](image)

- Ensure consistency in clock frequencies
  - Clock frequency checking module
  - Dynamically calculate frequency at run time

- Define coverage using SVA covergroups:

```
covergroup cg_cdc_gray_coded @(posedge tx_clock);
  cp_all_one_to_zero : coverpoint &one_to_zero_transitions     { bins cpb_one_to_zero_transitions = {1'b1}; }
  cp_values_checked  : coverpoint incr_values_checked                { bins cpb_values_checked = {1'b1}; }
endgroup: cg_cdc_gray_coded
```

- Enhance debug techniques
  - Minimize data for debug using PLI integration to report only unique set of errors
  - Configuration file to load the relevant cone of influence signals in the simulator
  - Integration of protocol coverage with CDC coverage:

![Debug Techniques Diagram](image)

- Automatically generate formal run setup automatically that is consistent with the CDC analysis setup
  - Clocks, resets, constants, black boxes
  - Use more accurate assertions that handle asynchronous data and clock transitions:

```
property data_stable(data, rx_clock, reset, areset);
  @posedge rx_clock disable iff(areset)
  !reset && $changed(data) |=> $stable(data)[*2];
endproperty
```

### Case Study: Simulation
- Used a set of SoC designs ranging from 1 to 100 million gates and containing 20 to 240 clocks.
- Comparison of simulation results:

<table>
<thead>
<tr>
<th>Existing flow and methods</th>
<th>Proposed techniques applied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error Frequency (%)</td>
<td>Error Frequency (%)</td>
</tr>
<tr>
<td>Test</td>
<td>Analysis</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
</tr>
</tbody>
</table>

- Significant reduction in assertion errors observed with enable condition in the assertion
- Run time improvement observed due to less number of assertion errors with PLI integration

### Case Study: Formal
- Used a set of SoC designs ranging from 1 to 100 million gates and containing 20 to 240 clocks.
- Comparison of formal verification results (number of assertions):

<table>
<thead>
<tr>
<th>User defined setup</th>
<th>Automatically generated setup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error Frequency (%)</td>
<td>Error Frequency (%)</td>
</tr>
<tr>
<td>Test</td>
<td>Analysis</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
</tr>
</tbody>
</table>

- Less iterations required to constraint formal verification tool to get conclusive results with the generated setup
- Reduced number of inconclusive assertions initial runs

### Conclusion
- The proposed techniques significantly improve the functional verification closure time of CDC paths
- Simulation performance is improved and the quality of verification is assessed with advanced coverage metrics
- Formal verification setup time is reduced through automatic generation of setup from CDC analysis
- The turnaround time of simulation and formal debug is minimized by the integration of formal debug with CDC results