Robust Vera Coding Techniques for Gate-Level and Tester-Compliant SoC Verification Environments

Mark Litterick, Verilab Ltd.
Joachim Geishauser, Motorola GmbH.
Outline

• Tester-compliant gate-level simulations have special requirements
  – for reliable operation Vera code must take these into account
• Consider why gate-level and tester-compliant simulations required
• Analyze resulting problems and restrictions
• Outline the effect on Vera operation
• Derive guidelines from analysis of connectivity and timing issues
• Apply guidelines to improve example code
• Conclusion
Why Gate-Level?

- Validate static-timing analysis
- Dynamic verification of selected paths
- Verify reset operation without RTL X-filtering
Why Tester-Compliant?

- Fault coverage requires normal-mode functional patterns
- Device characterization and parametric test
- Functional patterns required for automotive applications
• All restrictions apply to module-level Vera reused in SoC
Effect on Vera Operation

- Different behavior between RTL and gate-level simulations
- Different behavior between best-case and worst-case simulations
- Protocol and bus integrity errors when monitoring internal signals
- Compilation errors when connecting to gate-level netlist
- Timing violations in gate-level simulations
- Tester time-set mismatches due to uncontrolled transitions
All interfaces must connect to an appropriate clock
All interface definitions should explicitly declare a clock
Don’t mix internal and external interfaces in same definition
Vera Sync and Drive

- Don’t synchronize using async
- Don’t drive using async
Vera Sampling

- Don’t sample using async
- Don’t sample in expressions
- Don’t define or default to input skew of #0
Example Vera Code
(not tester or gate compliant)

interface mix_if {
    input int NSAMPLE #-1 hdl_node "tb.top.module.interrupt";
    input [3:0] bus NSAMPLE #-1 hdl_node "tb.top.module.bus";
    output rx PHOLD #1 hdl_node "tb.top.rx_pad";
}

bind mix_port mix_bind {
    int mix_if.int;
    bus mix_if.bus;
    rx mix_if.rx;
}

// Verification IP with virtual port (mix_port p)
@ (posedge p.$int async);
p.$rx = 1;
repeat(5) @ (posedge CLOCK);

if (p.$bus == 4'b0000) p.$rx = 0;

No Interface Clock Definition
Wrong Edge For Signals (Synthesized to posedge)
Mixed Internal and External Interface
Async Sample
Sync To Wrong Clock For Subsequent Sample
Async Sample In Expression
Alternative Vera Code (tester and gate compliant)

```verilog
interface internal_if {
    input int PSAMPLE #1 hdl_node "tb.top.module.interrupt";
    input [3:0] bus PSAMPLE #1 hdl_node "tb.top.module.bus";
    input clk CLOCK hdl_node "tb.top.module.clk";
}

interface external_if {
    output rx PHOLD #1 hdl_node "tb.top.rx_pad";
    input clk CLOCK hdl_node "tb.SystemClock";
}

bind mix_port mix_bind {
    int internal_if.int;
    bus internal_if.bus;
    rx external_if.rx;
}

// Verification IP with virtual port (mix_port p)
@(posedge p.$int);
p.$rx = 1;
@5 p.$bus == void;
var = p.$bus;
if (var == 4'b0000) p.$rx = 0;
```

- **Appropriate Interface Edge**
- **Explicit Interface Clocks**
- **Separate Internal and External Interfaces**
- **Minimal Changes to Bind**
- **No Changes to Port**
- **Synchronous Sample**
- **Implicit Delay Using Appropriate Clock**
- **Synchronous Sample**
Conclusion

• Derived guidelines for tester-compliant gate-level coding
  – ensure robust Vera code development
  – maximize reuse of module-level RTL code in SoC environment
  – apply to code reviews to assess reliability and reusability
  – rework code to fix reliability issues during debug

• Achieved identical operation for RTL and all gate-level conditions
  – functional test pattern generation and debug from RTL
  – validated static-timing analysis, selected timing paths and reset operation

• Achieved fault-coverage targets by supplementing ATPG and BIST
  – normal operation of logic modified or bypassed by DFT
  – any SoC simulation can be converted to a test pattern