Using SystemVerilog Assertions in Gate-Level Verification Environments

Mark Litterick (Verification Consultant)

mark.litterick@verilab.com
Introduction

• Gate-level simulations
  – why bother?
  – methodology

• Overview of ABV
  – applicability to gate-level simulations

• Reliable reuse of SVA from RTL to gate-level
  – clock distribution and negative hold-time

• Dedicated assertions for gate-level
  – timing checks and coverage
  – dynamically controlling SVA

• Comment on synthesizable assertions
Gate-Level Simulations

• SoC features => gate-level supplements RTL simulations
  complex clock relationships  elaborate power management
  dynamic frequency scaling    multi-voltage islands
  asynchronous operation       functional test patterns

• Gate-level simulations are used to:
  validate static timing analysis  detect dynamic timing defects
  check critical timing paths     verify reset operation (no X filter)

• Gate-level simulations do *not* replace STA or prove synthesis

• Apply appropriate methodology to maximize return-on-effort
  – identify features for gate-level validation in verification plan
  – specify appropriate functional coverage
  – target these features with intentional and stress test scenarios
Assertion-Based Verification

**Assertion-Based Verification** is a *methodology* for improving the *effectiveness* of a *verification* environment
- define properties that specify expected behavior of design
- check property assertions by simulation or formal analysis

**Benefits of ABV include:**
- improved error detection and reduced debug time due to observability
- improved integration due to built-in self-checking
- improved specification and documentation

**ABV considerations for gate-level simulations include:**
- reliable *reuse* of assertions from RTL environment
- additional assertions *dedicated* to gate-level
- *synthesizable* assertions
Reusing Assertions in Gate-Level Simulations

- When a verification environment depends on assertion-based checkers:
  - \( \Rightarrow relevant \) assertions must continue to operate at gate-level
  - otherwise benefits are lost
  - level of checking is reduced (missed defects)
  - failing simulations difficult to debug (lack of observability)

- Assertions must continue to work reliably at gate-level
  - no false failures (wasted debug effort)
  - no false passes (missed defects)

- Main gate-level attributes that affect assertion reliability:
  - clock-tree distribution
  - negative hold-time
RTL
Clock Distribution

- RTL clock is single net
  - for each clock domain
  - with no timing delays
- SVA sampling relative to any part of clock net is reliable
  - typically module CLK port
Gate-Level Clock-Tree Distribution

- Gate-level clock is distributed network of buffers/inverters
  - clock-tree balanced at registers
  - CLK port not on terminus
  - may be multiple clock ports
- SVA sampling must be relative to balanced clock
  - *not* module CLK port
  - typically a register clock input
Clock Connection Example

property p_count;
  @(posedge clk)
  (!$stable(count) |->
      (count == $past(count)+1'b1);
endproperty
bind tb.top.mod sva_chk sva_chk_i (  
  .clk    (clk),       // module port  
  .count  (count)
);

bind tb.top.mod sva_chk sva_chk_i (  
  `ifdef RTL //only: not appropriate for gate  
    .clk     (clk),        // module port  
  `else // GATE only: not present in RTL  
    .clk     (register.CK),  // register clock 
  `endif  
  .count  (count)
);

• Use conditional binding to appropriate clock for gate-level
Negative Hold-Time

• Normal flip-flop model
  – data setup-time prior to clock
  – data hold-time after clock

• Negative hold-time
  – allows data intended for the next cycle to change prior to clock edge
  – allows for increased clock skew
  – eases clock-tree synthesis
SVA with Negative Hold-Time

- Problem: SVA sampling in preponed region (value just before clock event) is not reliable in presence of negative hold-times

property p_nht_ex;
@ (posedge CLK)
$rose (SIG) |->
   (BUS=='next) && ($past (BUS)=='current);
endproperty

a_nht_ex : assert property (p_nht_ex);
SVA in Clocking Block

- SVA must operate within the scope of a *clocking block* for negative hold-times
  - Input skew must be greater than hold-time value
  - Skew of #0 or #1step do not work reliably
  - Input skew must not exceed setup-time value
  - Works for RTL and gate-level

```verilog
clocking ck_nht @(posedge CLK);
input #50ps SIG, BUS;
property p_nht_ex;
  $rose(SIG) ->
    (BUS==`next) && ($past(BUS)==`current);
endproperty
endclocking
a_nht_ex : assert property (p_nht_ex);
```
Dedicated Assertions for Gate-Level Simulation

- Features targeted for Gate-Level validation may not be fully addressed by RTL assertions and coverage
  - add new assertions for both RTL and Gate-Level
  - add dedicated assertions for Gate-Level only
- Dedicated assertions can be:
  - regular cycle-based properties
  - non-cycle-based timing checks
- SVA can be used to implement timing checks
  - timing checks can be used for assertions and coverage
  - under control of verification environment
  - **supplements** static timing analysis and structural timing checks in specify blocks, does not replace them
Timing Checks

• Gate-level models have timing checks built-in
  – system tasks in specify block (Verilog, SystemVerilog) or VITAL (VHDL)

• Not all timing defects trigger existing checks; for example:
  – no glitch present in best-case simulation
  – narrow pulse in worst-case, but *not narrow enough to violate* $width$
  – probable glitch in real-world (between best-case and worst-case)
SVA Timing Checks

- SVA allows *time* and *EventExpression* formal arguments
- Actual arguments can be static values or dynamic variables
- Can be used for:
  - pulse width violations (e.g. for clock quality checks at multiplexer)
  - period checks (e.g. for dynamic MVI protocol)
  - etc.

```verilog
property p_min_time(start, stop, duration);
time start_time;
 @(start)
 (1, start_time = $time) |=>
 @(stop)
 ($time - start_time) >= duration);
endproperty

property p_min_high;
 p_min_time(posedge clk, negedge clk, 2ns);
endproperty
a_min_high : assert property (p_min_high);

time minp = 6ns;
property p_min_period;
 p_min_time(posedge clk, posedge clk, minp);
endproperty
a_min_period : assert property (p_min_period);
```
SVA Timing Coverage

- SVA timing checks can also be used for functional coverage
- cover property statement records a *hit* if property evaluates to *true*
- Can be used to ensure gate-level environment created the required timing relationships

```
property p_max_time(start, stop, duration);
    time start_time;
    @(start)
        (1, start_time = $time) |=>
        @(stop)
            (($time - start_time) < duration);
endproperty
```

```
property p_max_time(data, posedge clk, 30ps);
endproperty
```

```
property p_just_before;
    p_max_time(data, posedge clk, 30ps);
endproperty
```

```
property p_just_after;
    p_max_time(posedge clk, data, 30ps);
endproperty
```

```
c_jb : cover property (p_just_before);
c_ja : cover property (p_just_after);
```
Performance and Reliability

• For performance and reliability reasons normally avoid:
  – assertions which trigger every clock cycle
  – non-synchronous assertions

• Performance overhead is much less in gate-level than RTL
  – many more simulation events occurring anyway
  – additional performance degradation not noticeable
  – care still required with coverage definition in particular

• Timing properties are prone to false failures
  – especially at start of simulation
  – dynamically control assertions and coverage if required
  – enable assertions and coverage only when safe
  – beware of over-constraining and missing defects at critical times
Dynamically Controlling SVA Timing Checks

- Named assertion and coverage statements controlled by:
  - `$asserton`, `$assertoff` and `$assertkill` system tasks
  - `disable iff` clause
  - `disable` statement

- Example:
  - controlling checks on internally generated launch-and-capture clock for at-speed transition fault scan tests
Controlling SVA using $asserton and $assertoff

property p_min_high; // as before...
a_min_high : assert property (p_min_high)
    else $error("%m: width violation on positive pulse");
initial $assertoff (a_min_high);

property p_pre_launch;
    @(posedge clk) $fell(scan_enable) |-> scan_mode;
endproperty

property p_post_capture;
    @(posedge clk) $rose(scan_enable) |-> scan_mode;
endproperty

a_pre_launch : assert property (p_pre_launch)
    $asserton(a_min_high);
    else $error("%m: fall on scan_enable when not in scan_mode");
a_post_capture : assert property (p_post_capture)
    $assertoff(a_min_high);
    else $error("%m: rise on scan_enable when not in scan_mode");
Synthesizable Assertions

- ABV can be extended into hardware-assisted verification flows
  - acceleration
  - emulation
  - FPGA prototyping
- Restricting assertion coding to synthesizable subset enables
  - embed assertion logic in physical netlist
  - synthesize part of testbench into target hardware
- Different scenario to ensuring testbench environment works with both RTL and gate-level representations of DUT
  - mentioned for completeness only...
  - ...not the focus for this paper!
Conclusion

• Benefits of ABV can be extended into gate-level verification
  – apply appropriate methodology to target key features

• SVA assertions can be reliably reused from RTL to gate-level
  – careful connectivity to terminal points of clock-tree
  – use clocking block with appropriate input skew

• Additional assertions can be specified for gate-level
  – functional and timing checks
  – additional coverage points for key features

• Improve effectiveness of verification environment
  – detect functional failures due to incorrect STA constraints
  – detect and isolate dynamic timing defects