Using SystemVerilog Assertions in Gate-Level Verification Environments

Mark Litterick, Verilab, Munich, Germany. (mark.litterick@verilab.com)

Abstract

Real-world requirements such as multiple clock domains and low-power modes of operation, including frequency and voltage scaling, often necessitate gate-level System-on-Chip (SoC) verification environments to complement the standard RTL based simulations. If the verification environment relies on assertion-based checkers to validate grey-box operation then gate-level simulations will also benefit from reuse of these assertions. This paper provides an overview of the problems related to gate-level timing and connectivity with a focus on how these affect the operation of the SVA code. A methodology is presented that enables reliable operation and reuse of the SVA in simulation environments that support RTL and gate-level implementations of the device-under-test. The paper also discusses SVA timing checks that can be added to the gate-level environment to improve checking and coverage.

1 Introduction

In order to maximize the benefit of an assertion-based verification methodology in a simulation environment that supports both RTL and gate-level implementations of the device under test (DUT), SystemVerilog Assertions (SVA) [1] must take account of gate-level connectivity and timing issues.

This paper provides an overview of the generic gate-level issues which directly affect the operation of properties and assertions. A methodology is presented that enables successful reuse of assertions (from the RTL environment) and supports enhanced checking of specific functional features targeted for validation by back-annotated gate-level simulations. The paper then discusses supplementary timing checks written in SVA that can be used to enhance the quality of the gate-level checking and functional coverage. Example SystemVerilog code implementations are provided to illustrate the solutions and restrictions.

2 Gate-Level Simulations

SoC designs with features such as complex clock relationships and elaborate power management (including dynamic frequency scaling and multi-voltage islands) often necessitate gate-level simulations to supplement the RTL verification environment [2]. In such simulation environments the RTL model for the device is replaced with a gate-level netlist using back-annotated timing information which is extracted after tasks such as synthesis, clock-tree insertion, scan insertion and place-and-route are complete. Gate-level simulations are used to:

- Validate Static Timing Analysis (STA)
- Verify critical path timing
- Verify reset operation without RTL filtering
- Detect dynamic timing defects

Gate-level simulations should not be used to replace STA, by attempting to dynamically validate all timing paths in the design. Nor should gate-level simulations be used to check for correct synthesis; that role is performed by equivalence checking. Where gate-level simulations are deemed necessary it is important to apply an appropriate methodology to maximize the return on effort:

- Identify features targeted for gate-level validation in the verification plan
- Identify corresponding functional coverage points and supplement if required
- Hit these features hard during simulation with intentional and stress test scenarios
- Do (only) a few smoke tests for normal operation
- Only use accurate post clock-tree synthesis timing

3 Role of Assertions

Assertion-Based Verification (ABV) is a methodology for improving the effectiveness of a verification environment. ABV involves the definition of properties, which specify the expected behavior of the design, and checking the assertion of these properties by simulation or formal analysis. Assertions are used to clarify specification

requirements, capture design intent and to validate correct operation and usage of the design. The benefits of ABV include improved error detection and reduced debug time due to increased observability. Refer to [3] for more details on assertion-based design and verification methodologies.

In this analysis we are mainly concerned with assertion-based checkers that monitor grey-box inter-module interfaces; however the requirements for assertions embedded in the RTL source are also discussed. In the context of gate-level simulations there are three main roles of assertions that need to be considered:

- reuse of assertions from RTL verification environment
- additional assertions dedicated to gate-level
- synthesizable assertions

3.1 Reusing Assertions in Gate-Level

If a verification environment relies on assertion-based checkers to validate internal operation or measure functional coverage then it is important that these assertions continue to operate if gate-level simulations are required. Otherwise the benefits of ABV are lost and corresponding level of checking during gate-level simulations is much less than the original RTL level environment - which means defects could go un-noticed or be difficult to debug. We must ensure that these assertions operate reliably when reused in the gate-level environment; in particular there must be no false failures or missed defects due to signal integrity and timing issues.

Many of the connectivity and timing problems detailed in [2] are still relevant for gate-level SystemVerilog simulations in general; however most are not an issue for SystemVerilog Assertions since they tend to be synchronous and passive. The most important gate-level attributes which directly affect the reliability of assertions are *clock-tree distribution* and *negative hold-time*.

3.1.1 Clock-Tree Distribution

A gate-level netlist has the clock net replaced with a buffer-tree structure during a back-end process called clock-tree synthesis. The clock tree has many buffers (and/or invertors) with the drive-strength and fanout loads balanced such that the clock edges arriving at each flip-flip, within the same clock domain, have an acceptable clock-skew relative to one another. A side effect of clock-tree synthesis is that the gate-level module hierarchy will typically have multiple clock ports, one of which will retain the original port name, but none of these clock ports will be on the terminus of the clock tree as shown in **Figure 1**.

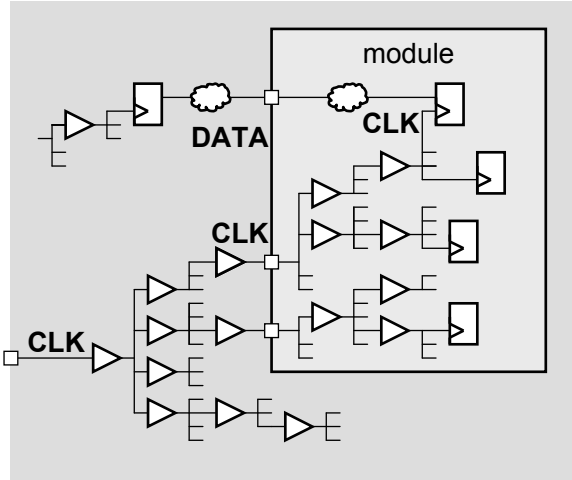

Figure 1: Clock Tree Distribution

Since data signals are only guaranteed to meet the setup time for the next clock edge at the balanced terminal points of the clock tree, assertions that use the module clock port can have unreliable sampling as shown in **Figure 2**. As a result these assertions can indicate false failures or passes due to incorrect sequence timing or corrupted data integrity.

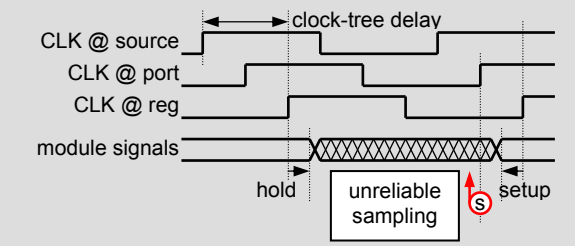

Figure 2: Incorrect Sampling Using Port Clock

An example of the type of SVA property that could generate a false failure under these conditions is shown in **Figure 3**. In this case *count* is expected to increment whenever it changes; however, since some of the bits have changed to the new value and some have not when the value is sampled, the assertion fails.

```
property p_count;
  @(posedge clk)
    !$stable(count) |->
      (count == $past(count)+1'b1);
endproperty
```

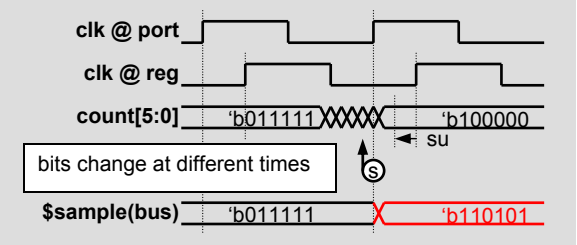

Figure 3: False Assertion Failure

An example of the type of SVA property that could generate a false pass under these conditions is shown in **Figure 4**. In this case *sig_b* is meant to be high when *sig_a* is high; however, the defective *sig_b* is one clock late, but since the assertion incorrectly samples the value of *sig_a*, the assertion passes.

```
property p_ab;
  @(posedge clk)
    sig_a |-> sig_b;
endproperty
```

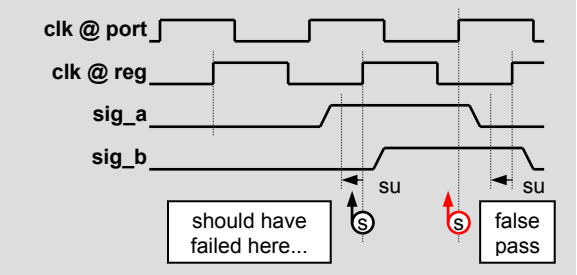

Figure 4: False Assertion Pass

For reliable operation the clock used for the SVA must be connected to a terminal point of the clock tree. For SVA that is connected to the DUT using the *bind* statement this can be achieved by connecting to the clock port of a register in the appropriate clock domain as shown in **Figure 5**. Note that this connection is conditional (based on a *`define RTL* in this example) since the register does not exist in the RTL netlist.

```
bind tb.top.mod sva_chk sva_chk_inst (
`ifdef RTL
  .clk (clk),
`else // GATE
  .clk (register.CK),
`endif
  .sig (sig),
  .bus (bus));
```

Figure 5: Conditional Binding to Clock

For SVA that is coded in-line with the RTL; it is reasonable to expect SVA-aware synthesis tools to (optionally) preserve the assertion in the generated gate-level netlist and to ensure that the clock is connected to the terminus of the clock tree whilst automatically preserving all other assertion signals in the netlist. Downstream tools which read the netlist would also have to recognize the SVA constructs and ignore them.

3.1.2 Negative Hold-Time

In a normal model for a flip-flop the data is expected to be stable some time before the active clock edge (the setup-time) and held stable until some time after this clock edge (the hold-time) as shown in **Figure 6(a)**. Most target libraries support the concept of Negative Hold-Time (NHT); which allows data intended for the next clock cycle to change prior to the current clock edge as shown in **Figure 6(b)**.

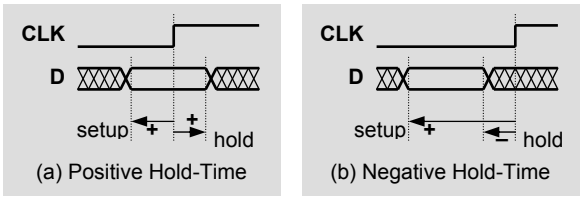

Figure 6: Hold-Time

Negative hold-times ease the task of clock-tree synthesis since they enable reliable operation with increased clock-skew. The output from a flip-flop with an early clock, intended for the next clock cycle, is allowed to arrive at the input to a flip-flop with a late clock prior to the clock edge without violating the timing (provided it arrives after the negative hold-time) as shown in **Figure 7**.

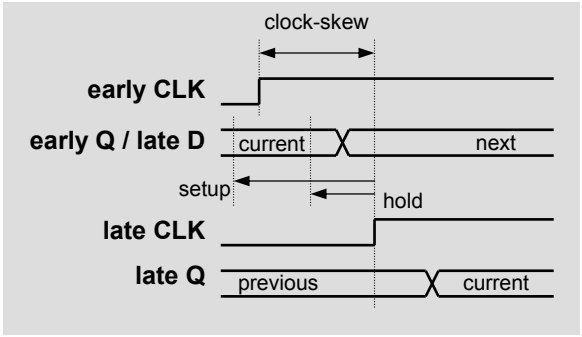

Figure 7: Clock-Skew with NHT

The problem for SVA in the context of back-annotated gate-level simulations with negative hold-times is that *sampling in the preponed region is not reliable*. **Figure 8** illustrates the problem, which

typically occurs under best-case timing conditions; in this case the 1-bit *SIG* and some of the N-bit *BUS* bits transition after the negative hold-time but before the rising clock edge, which is *legal* timing. If the SVA samples in the preponed region, just before the clock edge, then it would detect the transition on the single bit signal at the wrong time (one clock too early) and data integrity would be wrong for the bus value (since some bits change fast and others more slowly).

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

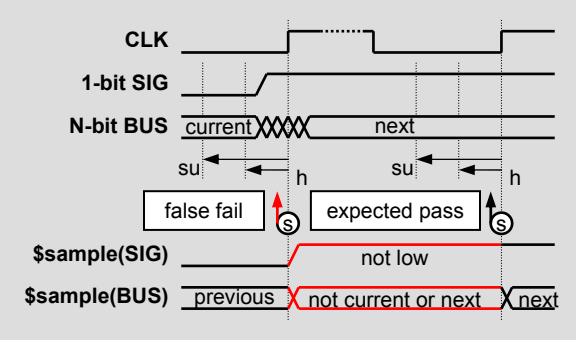

Figure 8: Preponed Sampling with NHT

The SystemVerilog mechanism for reliable sampling under these conditions is called the clocking block. A clocking block is a set of signals synchronized to a particular clock and it allows (amongst other things) the input sampling skew to be specified.

In the presence of negative hold-times, *SVA must operate within the scope of a clocking block where input skew values are greater than the negative hold-time*, but less than the setup-time, for the target library as shown in **Figure 9**. Clocking blocks with input skew values of *#0* or the default value of *#1step* are also unreliable. Note that although the input-skew is technology dependant, the clocking block and corresponding SystemVerilog operates reliably for both RTL and gate-level simulations.

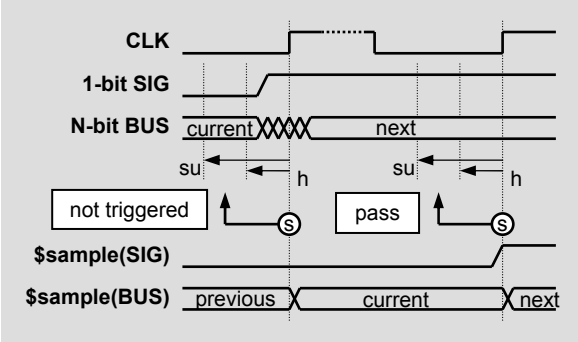

Figure 9: Clocking Block Sampling with NHT

There are a number of different ways to code using clocking blocks; one possibility is shown in **Figure 10**, where the property is defined inside the clocking block. In this case both *SIG* and *BUS* are sampled *50ps* before the rising edge of *CLK*.

```
clocking ck_nht @(posedge CLK);
  input #50ps SIG, BUS;

  property p_nht_example;
    $rose(SIG) |->
      (BUS==`next) && ($past(BUS)==`current);
  endproperty

endclocking
```

Figure 10: SVA in Clocking Block

Note that one of the restrictions of coding assertions inside a clocking block is that multiple-clocked assertions are not supported.

It is probably too much to expect SVA-aware synthesis tools to infer and connect a clocking block for you; however it is reasonable to expect any clocking block constructs and associated assertions to be preserved into the gate-level netlist.

3.2 Dedicated Assertions for Gate-Level

It is possible that the features targeted by gate-level simulations in an assertion-based verification flow are not well addressed by existing assertions and coverage from the RTL part of the flow. In such cases it may be appropriate to define additional assertions and coverage which are only used during the gate-level simulations.

As well as additional cycle-based functional assertions for the gate-level environment it is also possible to use SVA to perform non-cycle-based timing checks as illustrated by the following example. In **Figure 11** a defective clock multiplexer operates correctly during both RTL and best-case

gate-level simulations; however, under worst-case conditions an illegal narrow pulse is present but it is sufficiently wide that it *does not violate the timing checks* on the gate-level flip-flop model. The problem is that real-world operation will be somewhere between best and worst-case and it is quite likely that this circuit will generate illegal narrow pulses on the clock network during real-life operation. It is possible that an intermediate gate-level condition (typical or tester) would cause a timing check fail, but not guaranteed.

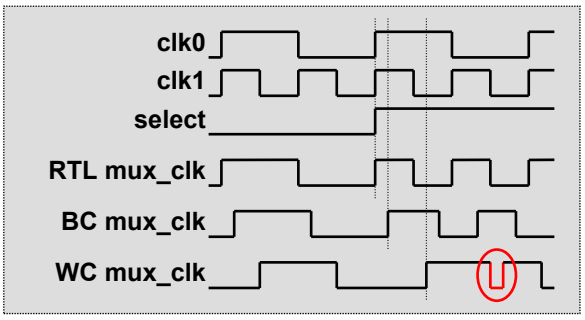

Figure 11: Undetected Clock Glitch

One possible solution to detect such dynamic timing defects is to add additional HDL timing checks to the clock signal using Verilog system tasks in a specify block (e.g. $width) or corresponding VITAL checks for VHDL; however it is also possible to perform these checks using SVA, which has the advantage of containing the checks in the verification environment. **Figure 12** illustrates two generic multiple-clock SVA properties which can be used for timing checks; these property definitions are possible since SVA allows *EventExpression* and *time* variables to be passed as formal arguments.

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

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

Figure 12: SVA Timing Check Properties

There are a number of possible applications of the properties shown in **Figure 12**, including detecting high level glitches and incorrect operating frequency for the current environmental conditions (for example based on dynamic voltage scaling), as shown in **Figure 13**. Notice that the complete *EventExpression* (e.g. *posedge clk*) is passed as the formal argument. The *p_min_high* property instantiates *p_min_time* to check that the minimum time between a rising and a subsequent falling edge is not violated. The *p_pin_period* property performs a similar check based on two successive rising edges of *clk*. Note also that the actual value for the duration argument can be a variable that changes during simulation; for example to model dynamic frequency scaling or voltage controlled power management.

```
property p_min_high;
  p_min_time(posedge clk, negedge clk, 10ps);
endproperty : p_min_high

time minp = 6ns;
property p_min_period;
  p_min_time(posedge clk, posedge clk, minp);
endproperty : p_min_period

a_min_high : assert property (p_min_high)
  else $error("...too narrow");

a_min_period : assert property (p_min_period)
  else $error("...too fast");
```

Figure 13: SVA Timing Assertions

The timing check properties from **Figure 12** can also be used to define functional coverage points using SVA cover statements. For example, when interaction between clock domains is targeted for gate-level simulation, we might want to ensure that the testbench environment actually created clock relationships which allowed key signals to change close to the destination clock. In this case the *p_max_time* property can be used as shown in **Figure 14**. Note that one of the *EventExpression*s in each property is *@(data)*, which means any transition of the *data* value. Since the coverage is recorded when the properties holds true, these cover points will only record a hit of the *data* transitions within *+/- 30ps* of the rising edge of the clock.

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

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

c_just_before : cover property (p_just_before);
c_just_after  : cover property (p_just_after);
```

Figure 14: SVA Timing Coverage

This approach is contrary to popular texts on assertion-based design, such as [3], which normally guard against using assertions for glitch detection, non-synchronous use of assertions and also assertions that potentially trigger every clock cycle. In general this advice is valid; however the assertions presented here have successfully been used to detect and isolate defects during gate-level simulation which would not otherwise have been observed. The important things to be aware of are simulation performance and false failures.

The first thing to note is that the performance overhead from having an assertion trigger on every clock edge is much less in gate-level, than it is in RTL simulations, since there are many more simulation events occurring anyway. In other words the performance is bad anyway - so a few assertions result in minimal additional performance degradation. Nonetheless care is required, particularly where coverage is concerned, and the verification engineers should be careful about how timing check assertions are used.

In order to avoid false failures, particularly at the start of simulation, it might be necessary to dynamically control the operation of such assertions and coverage using the *disable* statement, the *$asserton*, *$assertoff* and *$assertkill* system tasks, or the *disable iff* clause. Care should be exercised to avoid over constraining which could result in missed defects during critical phases of operation.

3.3 Synthesizable Assertions

Certain types of assertions can be extended into hardware-assisted verification flows such as acceleration, emulation and FPGA prototyping [4]. By restricting the assertion coding to a synthesizable subset of the language (in much the same way as RTL is a coded in a synthesizable subset of Verilog or VHDL) it is possible to embed assertion logic in the physical netlist for the DUT (or indeed synthesize part of the testbench environment into the target hardware). Note that the assertions physically become part of the netlist, i.e. the assertions are converted into gates, and therefore issues related to clock-tree synthesis and connectivity as well as support for negative-hold-times is guaranteed.

This is a different scenario to ensuring that a testbench environment operates correctly with both RTL and gate-level implementations of the DUT. It has been included here for completeness and clarity but is not the focus for this paper.

4 Conclusion

This paper has demonstrated that it is possible to reuse SystemVerilog Assertions in a gate-level simulation environment provided clocking blocks are used with appropriate input skew and care is taken with the clock connectivity. In addition the possible application of SVA to perform non-cycle-based timing checks and related functional coverage was presented. A combination of these techniques can be used to ensure that the benefits of assertion-based verification are extended to the gate-level simulation environment which helps to maximize the corresponding return on effort.

5 References

[1] Accellera, *SystemVerilog 3.1a Language Reference Manual*, http://www.accellera.org

[2] Litterick & Geishauser, *Robust Vera Coding Techniques for Gate-Level and Tester-Compliant SoC Verification Environments*, SNUG Europe 2004 & MTV 2004 [1]

[3] Foster, Krolnik & Lacey, *Assertion-Based Design 2nd Edition*, Kluwer Academic Publishers, ISBN: 1-4020-8027-1

[4] Newell & Schutten, *Bringing assertions into hardware-assisted verification*, EE Times 2004

[1] Available from: http://www.verilab.com/download.htm