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

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

mark.litterick@verilab.com
joachim.geishauser@motorola.com

ABSTRACT

Real world requirements such as low-power modes of operation and multiple clock domains often necessitate gate-level System-on-Chip (SoC) verification environments. Additional complexities introduced by tester compliance impose restrictions on the control and repeatability of simulations over all situations, including register-transfer-level (RTL) and different gate-level conditions. Making full use of the Vera high-level verification language in these circumstances requires special considerations and techniques not normally applied in a module-level RTL testbench. If the intent is to reuse Vera monitors, drivers and result-checkers in the gate-level SoC environment then the code must be designed appropriately. This paper first explores the generic issues of interacting with a gate-level SoC in a tester compliant manner and then proceeds to derive Vera coding guidelines that ensure robust operation across a range of testbench abstractions from module-level RTL through to tester-compliant gate-level SoC implementations.

1.0 Introduction

This paper is based on practical experience gained during the development of a Vera verification environment for a family of complex SoCs. The resources (effort, time and cost) required to rework notionally correct Vera code to perform reliably in the gate-level simulations was the catalyst for developing a methodology that would place minimal restrictions on the verification IP development and allow maximal reuse.

Outlining the types of features that may necessitate a gate-level simulation environment the paper also considers why functional test patterns are often required for complex SoCs. A discussion on relevant issues and restrictions imposed as we move up the device implementation and testbench abstraction levels from RTL through gate-level and tester-compliant SoC verification environments is presented.

By analyzing the effects of netlist generation (net renaming, clock-tree synthesis, etc.) and the requirements for tester compatible operation (functional pin timing, test vector generation rules, etc.) the paper highlights factors that have to be taken into account in the development of the testbench infrastructure and its components.

Focusing on connectivity and timing issues, guidelines are derived for the development of Vera verification IP and corresponding interfaces. These guidelines ensure reliable operation in the standalone RTL module-level testbench, and when reused at the system-level with both RTL and gate-level implementations of the SoC.

The paper contains the following sections:

- SoC Complexity
- Problem Space
- Analysis of Vera Interactions
    - Connecting to Signals
    - Synchronization of Signals
    - Driving Signals
    - Sampling Signals
- Conclusions and Recommendations
- Acknowledgements
- References
- Appendix: Vera Guidelines

2.0 SoC Complexity

Most of the analysis presented in this paper is based on general considerations for complex SoCs; however the catalyst for the paper and the driver for the analysis was the development of a Vera verification environment for the Motorola MAC7100 family of SoC devices. The SoC outlined in this section is representative of the level of complexity facing today's design and verification engineers.

2.1 SoC Overview

A simplified block diagram of a MAC7100 family SoC is given in Figure 1. It consists of an ARM7TDMI CPU, multiple bus hierarchy, on-chip memory, DMA, system support functions and many peripherals. Targeted at embedded automotive applications the peripheral set includes SCI, SPI, I^2C, and FlexCAN serial interfaces, A/D converters, timers and general-purpose I/O.

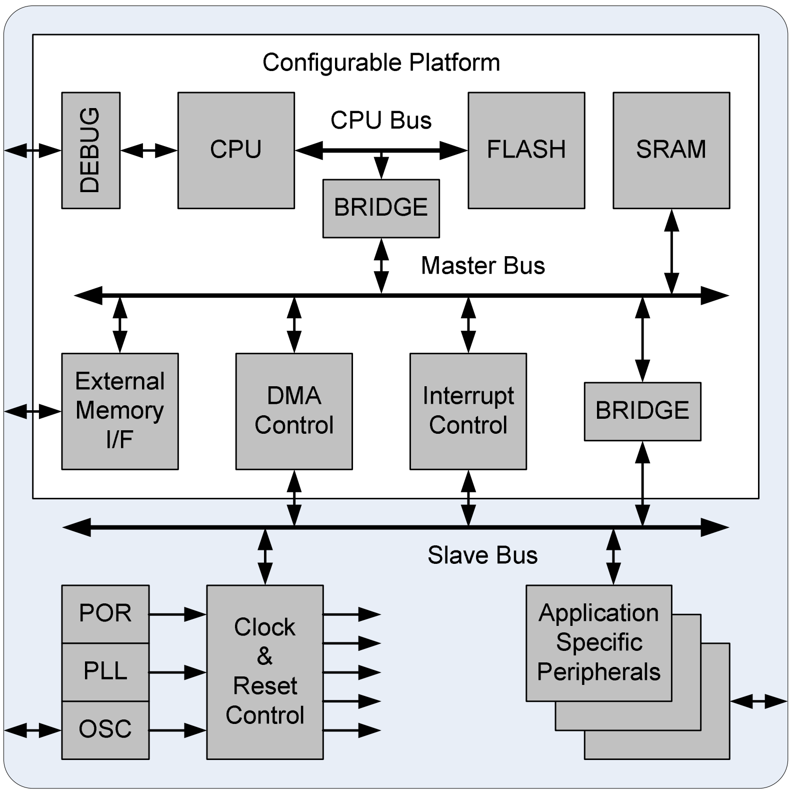

Figure 1: SoC Block Diagram

Aspects of the device architecture that directly affect the Vera verification environment include the following:

- Multiple clock domains
- Derived clock modes using PLL
- Peripheral slave bus operating from divided clock
- Power management with multiple gated clock branches
- Asynchronous wakeup from low-power modes using RC-oscillators
- Analog blocks including oscillator, power-on-reset and A/D converters

- Total delay from clock input to device outputs of more than one clock period
- Implementation library supports negative hold times
- DTF logic for SCAN and BIST interferes with clocks and modes

2.2 Verification Methodology

The verification methodology comprises the following stages:

- Extensive module-level verification including constrained-random and directed tests
- Bus-Functional Model (BFM) system-level constrained-random and directed tests
- Software-Driven Verification (SDV) system-level directed tests

The module-level test environments are mainly written in Vera and are used to fully validate the functional behavior of the module, or platform, independently from any SoC in which it is used. The module supplier is responsible for functional verification of the module. Most of the module-level simulations are performed on RTL only.

The BFM verification environment is written entirely in Vera and is used to fully test the integration of the modules in the SoC. These tests are responsible for validating the connectivity and interoperability of each module and the system as a whole. In the BFM testbench the CPU core is replaced by a stub and a Vera driver provides bus-functional procedures that are used to directly stimulate and respond to the SoC through the embedded interfaces. All stimuli for the BFM environment are written in Vera and the bus-functional modeling of the CPU enables much faster system-level operation than SDV. Both RTL and gate-level simulations can be performed in the BFM environment and since it does not have to be tester-compliant it is also capable of fully asynchronous clock domain interaction.

The main purpose of the SDV environment is to fully validate the SoC operation with the embedded CPU core executing representative software in a normal manner. The SDV environment is written in Vera with additional software stimulus written in ‘C’. The SDV environment is fully tester-compliant and therefore any of the corresponding tests can be processed into test patterns and applied to real silicon. Both RTL and gate-level simulations can be performed using the SDV environment.

2.3 Why Gate-Level?

The complexity of the timing relationships resulting from features such as multiple clock domains, derived clocks, low-power modes and analog blocks makes it difficult to express corresponding timing constraints accurately and apply them correctly during static-timing-analysis (STA). In such cases relying purely on RTL simulations supplemented by STA may be inadequate and it is generally accepted that gate-level simulations remain a necessary evil in most current SoC design flows [1]. The gate-level simulations are used to validate static-timing analysis and not as a means of dynamically verifying all the timing paths in the design. Gate-level simulations are also capable of detecting bugs with reset operation that may not be visible in RTL due to filtering effects in HDL coding.

2.4 Why Tester-Compliant?

The same design features that necessitate timing validation by gate-level simulation also require special consideration when it comes to manufacturing test. Specifically many of the low-power, multiple-clock domain and analog features will be modified or bypassed by design-for-test (DFT) logic during SCAN or BIST operation. High reliability applications, such as automotive, demand high fault coverage and require supplementary functional test patterns to exercise the normal operational mode of the logic that has been modified or bypassed by the DFT logic. Additional reasons for functional test pattern generation include device characterization and parametric testing of features such as run, doze and stop IDD values. For complex SoCs the risk of test issues arising on the actual tester is relatively high if the source simulations are not compatible with tester operation and test vector processing tools. All of the simulations that are converted to test patterns for execution on a real device are performed on both RTL and gate-level SoC implementations.

2.5 Why Reuse?

Reusing module-level verification code, such as drivers, monitors and response-checkers, in the SoC environment enables efficient checking of the system connectivity and interoperation as well as validation of the environment in which the module is instantiated. When the module functionality is fully verified in a standalone testbench, but only partially exercised in the SoC environment, interface monitors and protocol checkers can be reused to validate real interface operation against the emulated interfaces in the module-level testbench and ensure that the module is not operating outside its pre-verified range of functionality.

3.0 Problem Space

The main consideration for this paper is how do we ensure that Vera code written for a module-level RTL testbench is robust enough to be used in a SoC verification environment that supports RTL, gate-level and fully tester-compliant simulations.

At the module-level there are almost no restrictions on the interaction of Vera with the DUT in terms of connectivity and timing; however as we consider the requirements for RTL, gate-level and tester-compliant SoC verification environments, progressively more restrictions affect the way Vera interacts with the DUT in its various forms. The following sections outline the types of restrictions that the different levels of SoC implementations bring.

3.1 RTL SoC Simulation Environment

By virtue of the fact that the module is now instantiated in a SoC the module-level verification code has to be capable of operating correctly even though:

- The interface clock(s) may no longer be closely related to the SoC testbench environments default *SystemClock* (e.g. the slave bus interface clock is *SystemClock*/2, or the SoC uses a PLL clock generator)
- The module interfaces may now be a mixture of internal interfaces (e.g. system bus) and external interfaces (e.g. device pins)

- Interface clocks may be gated by power-management logic and might not always be present
- The clock period in the SoC environment may be different
- The Vera timescale in SoC environment may be different

3.2 Gate-Level SoC Simulation Environment

Back-annotated timing simulations with a gate-level netlist introduce many additional complications for both connectivity and timing of the verification code.

Gate-level connectivity issues include:

- Clock nets are replaced with clock-tree structures
- High fan-out nets (like Reset and Test) are replaced with buffer-tree structures
- Internal RTL net names may not be preserved in hierarchical gate-level netlist
- Module port names are mostly preserved, but with some exceptions:
    - Port names may be modified by back-end tools
    - Port names and polarity may be modified due to clock path inversion
    - Unused bits of bus ports may be retained but unconnected

Gate-level timing issues include:

- Realistic timing delays on all signals need to be accounted for
- Timing changes with simulation conditions (best, typical, worst, etc.)
- Best-case must match worst-case to validate full dynamic timing range
- Gate-level should match RTL to allow for efficient debug
- Clock-trees have intrinsic skew (clocks on same domain have different timing)
- Clock-trees introduce large delay into clock path
- Clock mark-to-space ratio distortion due to different edge rates may affect timing
- Clock ports on modules may not be at terminus of balanced clock-tree
- Synthesis tends to maximize all worst-case path delays to just meet timing constraints with minimal area and power
- Timing violations must be avoided (by controlling timing) or managed (by disabling timing checks on the first stage of meta-stability correction logic)
- X values are propagated (unlike RTL, which filters them)
- Negative hold-times may be supported by implementation library

3.3 Tester-Compliant SoC Simulation Environment

A tester-compliant simulation environment is one in which the interaction of the testbench infrastructure with the DUT is such that the signal transitions at the device pins can be converted to test patterns that are compatible with the operational capabilities of automated-test-equipment (ATE, referred to as a *tester* in this paper). The tester is not a hardware version of a simulator [2] and is not capable of emulating every aspect of the verification environment. The tester has to contend with real world issues, such as:

- transmission line effects and delays
- signal degradation, noise and cross-talk

- large capacitive load on DUT pins
- signal timing inaccuracies, skew, jitter and edge-rates
- limited number of signal generators, power supplies and measurement instruments
- limited number of channels (pins)
- limited memory depth
- limited number of time-sets
- X is not a valid real-world state
- Z is only measured during dedicated high-impedance test

A tester time-set consists of timing information relative to the fundamental tester-cycle and a tester timing format, e.g. Return-to-Zero (RZ) or Non-Return-to-Zero (NRZ). Since every signal transition at the device pins is matched with a tester time-set by the pattern processing tools it is important to carefully control the signal transitions for the duration of each pattern in order to ensure that the overall number of time-sets is compatible with tester capability.

The main requirement for making a Vera testbench environment tester-compliant is the absolute control of all timing events at the device pins and the consistent application of that timing. Note that the interface timing belongs to the SoC environment in which the module is instantiated and not to the reusable verification IP. The interface specifications associated with external pins effectively define the corresponding time-set and provided this timing is carefully applied (for example using normal Vera synchronous coding) downstream pattern processing and subsequent pattern consistency can be ensured.

If functional patterns are required for SoC validation then the justification for making the simulation environment tester-compliant is to reduce cost. This is achieved by simplifying the test pattern generation flow (since minimal modifications happen to the signal transitions between simulator and tester), reducing back-end test program complexity and generation effort, and providing a more efficient and powerful debug environment with faster iteration loop. The cost of debugging test programs on real silicon is severe and typically impacts delivery of first silicon to the customer. An additional benefit of a testbench environment that closely emulates the tester is that the accuracy of the predicted fault coverage is improved.

It is possible to develop a tester-compliant RTL verification environment and postpone gate-level validation to a back-end re-simulation stage of the processed patterns that does not use the source testbench. We chose a strategy that allowed for tester-compliant operation of both gate-level and RTL simulations and evolved towards performing less gate-level simulations and generating test patterns from the RTL as the project matured. The advantages of this approach include:

- Capability of performing any test under all conditions
- Dynamic verification of complex timing scenarios for directed test cases
- Validation of static timing analysis and synthesis constraints
- Debugging tests is much easier in source environment (compared to the re-simulation testbench) since log messages, faster iteration loop and less processing stages
- Allowed functional test patterns to be derived from gate-level simulations long before all the Vera operational inconsistencies were resolved

The following restrictions apply to tester-compliant verification environments for both RTL and gate-level implementations of the DUT:

- All input stimulus must match consistent tester time-set(s)
- Limited number of tester time-sets are available
- All clock periods and delays must be related to tester cycle
- All stimulus events must be synchronous to tester cycle
- Stimulus is required to load internal memory on tester

Gate-level tester-compliant simulations include these additional restrictions:

- All output transitions must allow for consistent output strobe positions according to tester rules (e.g. no transitions within the last few nS of tester cycle)
- Timing violations (that have observable effects) must to be avoided (by controlling timing)
- Disabling timing checks on the first stage of meta-stability correction circuits is not allowed in tester-compliant modes
- Clock alignment may be necessary in testbench (e.g. to match *SystemClock* with DUT clock output from PLL or similar) and using clock search on tester
- Pattern synchronization may be necessary in testbench (e.g. after an asynchronous wake-up validated with an RC-oscillator) and using match-loop on tester
- Tester round-trip-delay needs to be accounted for during match-loop stimulus

Having presented a generic overview of the limitations and restrictions enforced by the different testbench abstractions the remainder of the paper is devoted to providing guidance on how to write robust Vera code that will operate reliably in all of these environments.

4.0 Analysis of Vera Interactions

This section provides a detailed analysis of the issues relating to connecting, synchronizing, driving and sampling using Vera in tester-compliant RTL and gate-level SoC verification environments with an aim of achieving reliable and robust operation throughout all simulation conditions. Readers are encouraged to refer to the Vera User's Manual [3] and The Art of Verification with Vera [4] for general information on Vera interface operation and coding requirements.

The analysis is targeted at determining how Vera code can be written to maximize reuse in both RTL and gate-level simulations with a particular focus on avoiding code constructs that prevent reliable operation in the SoC environment despite being perfectly valid in the module-level RTL verification environment.

The code examples presented in this paper assume that all connections are made using direct *hdl_node* notation in the interface definitions, static binding is performed using the *bind* construct and all signals are referenced using *virtual ports*.

The guidelines derived in this section are cross referenced to the Appendix which contains a summary of each guideline with code examples and additional information; these references are of the form [An].

4.1 Connecting to Signals

In order to illustrate the interface clock connection issues we will first consider the simplest case; that of the single clock-domain SoC. In such a case the *SystemClock* will typically be connected to the DUT clock input and the internal clock net will be replaced with a synthesized clock-tree in the gate-level netlist.

It is an attribute of synthesized designs that the worst-case path delays are maximized to just fit the available time budget. For external signals the total delay from the clock input to an output is guaranteed to meet the necessary setup constraints for another device in the system. Likewise internal signal paths will meet the setup time for the target flip-flop; however this will typically be after the external clock transition due to the transport delay of the clock-tree. The timing of internal and external signals is shown in Figure 2:

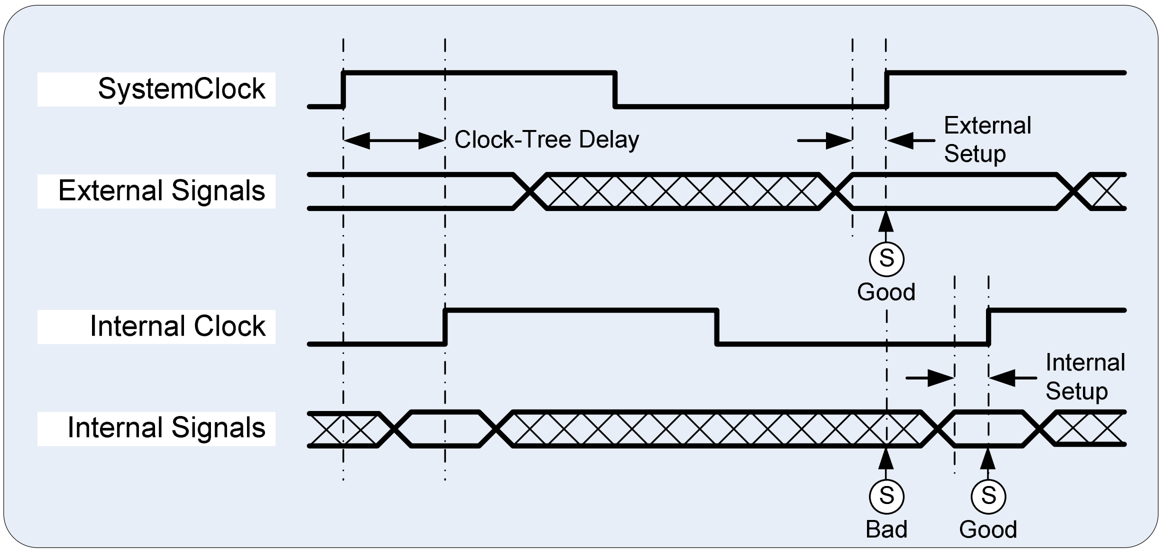

Figure 2: Internal and External Timing with Single Clock Domain Device

The cross-hatched values in Figure 2 represent the difference between the minimum and maximum path delays for all signals in the corresponding clock domains. A detailed discussion of sampling techniques is given in Section 4.4, for the moment consider the effects of sampling on or just before the corresponding clock edge – shown as (S). From Figure 2 it is clear that it is safe to sample external signals using *SystemClock* timing (provided *SystemClock* is connected to the device clock) but that *SystemClock* is not appropriate for sampling internal signals even in a single clock-domain device. If the internal signal being sampled is a single bit, then the worst case simulation may get the result one cycle later than the best-case and RTL simulations; however if Vera is sampling an internal bus then it is possible that the sampled value will be wrong if some of the bits happen to transition prior to the *SystemClock* edge and others transition after the edge.

A similar problem occurs when the SoC has an internal slave bus operating at half of the frequency of the main system bus as shown in Figure 3. Synthesis will ensure that logic on the slow clock domain utilizes the available path delay in order to reduce power and routing demands on parts of the SoC where performance is not an issue. In this case it is more obvious that the *SystemClock* is inappropriate for sampling internal signals on the divided-by-2 clock domain.

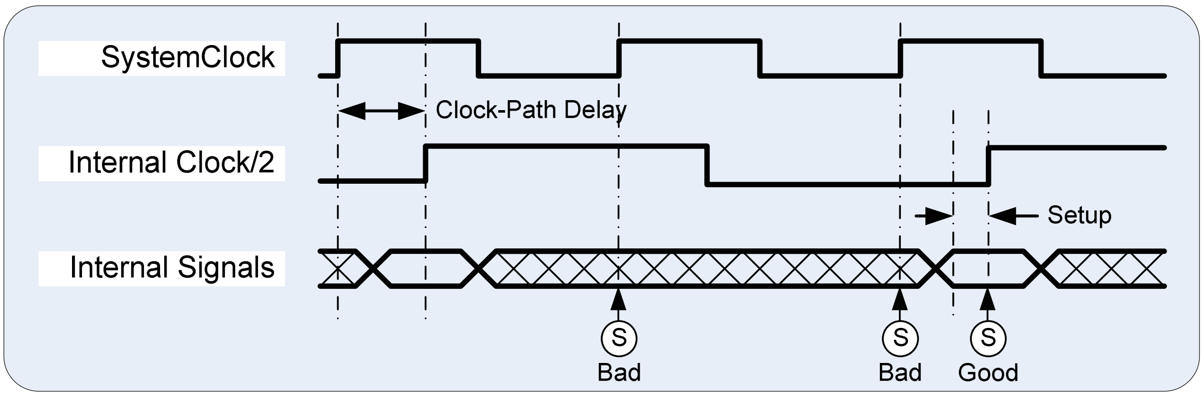

Figure 3: Divided Clock Timing

Without analyzing any more complicated examples it is possible to conclude a number of interface clock connection guidelines at this stage. Firstly, all interfaces must connect to an appropriate clock [A1]; in particular interfaces that are monitoring (or driving) internal signals must connect to an appropriate internal clock. We would recommend that all interface definitions should explicitly declare a clock [A2] and connect it to an appropriate clock signal, even if the appropriate clock is *SystemClock*, in order to explicitly identify the associated clock domain. If all interfaces connect to an appropriate clock it follows that you cannot mix external and internal signals in the same interface definition [A3], since they each need their own appropriate clock. Interfaces that were completely external in the module-level environment can become a mixture of internal interfaces (like the register bus) and external interfaces (like the data pins) in the SoC environment and therefore they require different interface definitions to handle the separate clock domains as shown in the following code example:

```
// Refactor the following mixed interface for module-level VIP...
interface mix_if {
  input clk  CLOCK              hdl_node "testbench.module.clk";
  input bus  PSAMPLE # -1       hdl_node "testbench.module.bus";
  input pin  PSAMPLE # -1       hdl_node "testbench.module.pin";}

// ...into separate interfaces for SoC operation
interface internal_if {
  input clk  CLOCK              hdl_node "testbench.soc.module.clk";
  input bus  PSAMPLE # -1       hdl_node "testbench.soc.module.bus";}
interface external_if {
  input clk  CLOCK              hdl_node "testbench.soc.clk";
  input pin  PSAMPLE # -1       hdl_node "testbench.soc.pin";}

// The port definition and VIP code are unchanged but the bind is modified:
bind mix_port mix_bind {
  bus internal_if.bus;
```

pin external_if.pin;}

Signal connectivity is the responsibility of the SoC testbench and not the module-level verification IP since the reuse environment and timing cannot be predicted. However the port definition is supplied with the verification IP and should permit appropriate SoC interfacing. Care is required if port definitions include clock signals and explicit synchronization is used.

Hierarchical synthesis techniques typically employed in SoC methodologies normally preserve the module port names but do not tend to preserve net names internal to the modules. Accordingly module-level verification IP that is intended to be reused in a gate-level SoC environment should interact only with module ports and not internal nets [A4].

Special care is required with internal clock nets in the gate-level netlist since the module port may not be the terminus of the balanced clock tree. Figure 4 illustrates a typical scenario that can develop in a netlist after the clock-tree synthesis stage with the corresponding timing shown in Figure 5:

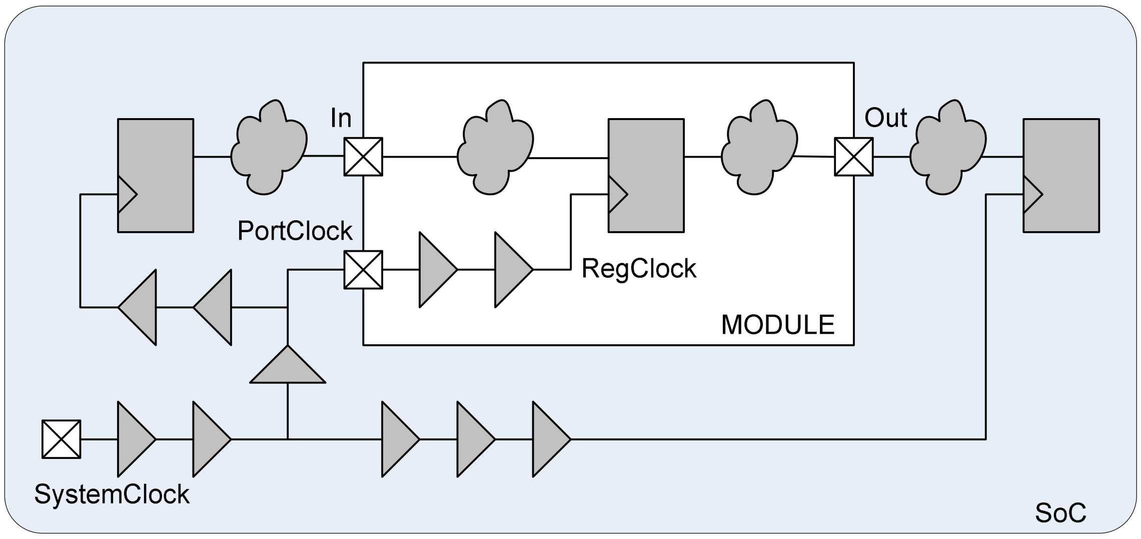

Figure 4: Clock-Tree Structure

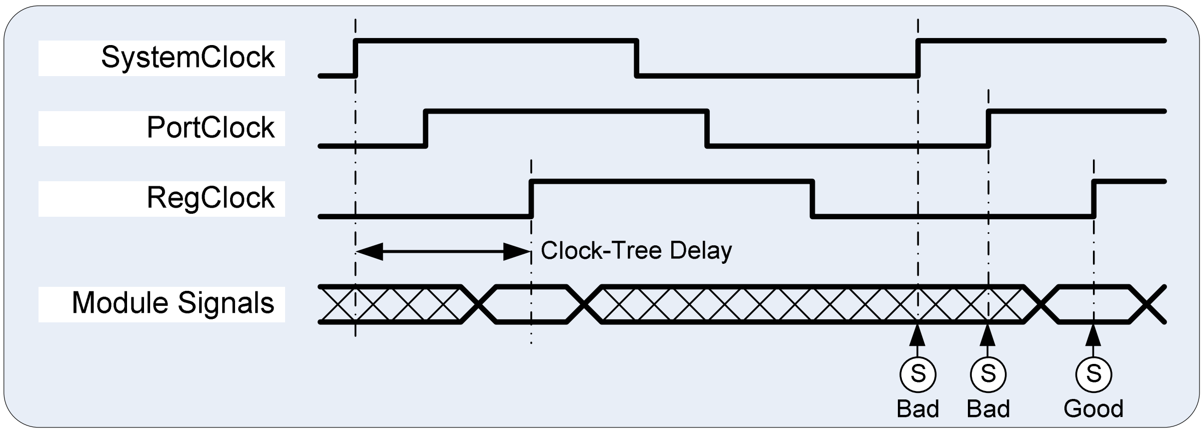

Figure 5: Clock-Tree Timing Diagram

In Figure 4 all three registers are in the same clock domain. Clock-tree synthesis ensures that the clock skew is within acceptable limits and there are five clock buffers in each path. The timing of the module input and output signals is effectively unknown but is guaranteed to meet the setup and hold timing for all other registers in the same clock domain. Verification code that is monitoring the module interface ports will operate unreliably if it is synchronized to either *SystemClock* or PortClock but will be robust when using RegClock as shown by the corresponding good and bad samples, (S), in Figure 5.

In fact clock-tree synthesis can also result in module clock ports being inverted and renamed in the netlist (e.g. clk is changed to clk_inv). In order to guarantee proper interface operation in circumstances like these the clock signal should be connected to any of the terminal points of the balanced clock-tree for the corresponding clock domain. If an appropriate clock port is not available then the safest alternative connection point is the clock input to a register in the corresponding clock domain. Unfortunately register does not exist in the pre-synthesis RTL design and so a conditional connection will have to be made depending on the current simulation conditions. Note that since register names are generally repeatable and reliable the corresponding interface connections tend to be independent of netlist release. The following code illustrates one possible implementation in the interface definition when the environment identifies the current simulation mode using a *plusarg* or similar:

```
#ifdef RTL
  input  clk  CLOCK  hdl_node “testbench.soc.module.PortClock”;
#else // GATE
  input  clk  CLOCK  hdl_node “testbench.soc.module.appropriate_reg.CK”;
#endif
```

One final gate-level connection issue is worth considering; when a multi-bit bus port is preserved in the netlist but not fully implemented in the design (e.g. if some of the bits are not required) the corresponding bus bits may be unconnected. Since the port physically exists in the netlist Vera connectivity issues will not be apparent at compile time but the verification code may exhibit erroneous behavior since some of the bus bit values will be *‘z’*. Be aware of this potential hazard when debugging the gate-level simulations and resolve it by connecting the appropriate interface bus bits to a constant (either for using conditional connections as before or just use the same connection for RTL and gate-level) as shown in the following example:

```
input  [15:0]  addr  PSAMPLE  # -1  hdl_node
                    “{tb.soc.module.addr[15:12], 4’h0, tb.soc.module.addr[7:0]}”
```

Gate-level compliant Vera connections typically become tied to specific releases of the netlist. Physical connectivity problems usually result in *cross-module resolution errors* when the testbench is compiled and are easy to spot although can be time-consuming to rectify. Timing related issues, like the clock port position in the tree, are much harder to predict and debug – in this case the best line of defense is an awareness that these problems may exist in the gate-level simulations and if failures do occur to go looking for an appropriate solution.

4.2 Synchronization of Signals

Vera provides the capability of synchronizing to any signal in the DUT. This section looks at the implications of both synchronous and asynchronous synchronization to data and clock signals in the context of both RTL and gate-level verification environments.

Synchronizing to a data signal transition is really concerned with detecting a change in level and not specifically aligning the Vera timing with the transition (the converse is true for clock signal transitions); however a side-effect of the short-hand notation for detecting a transition on a signal is that the timing can be affected. Consider the relative timing of a data signal transition under different simulation conditions shown in Figure 6:

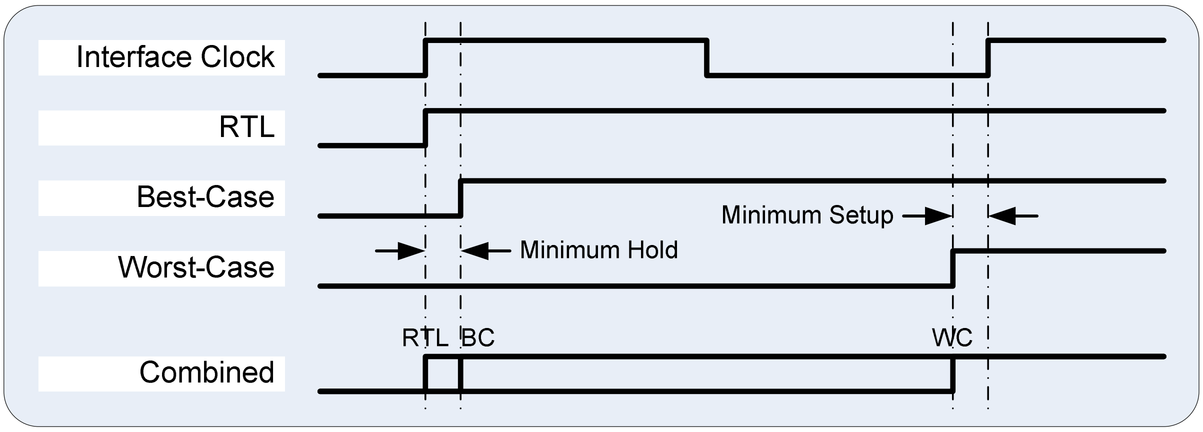

Figure 6: Data Signal Transitions for RTL, Best and Worst-Case

In Figure 6, the RTL signal transition occurs one simulator time-step delay after the clock edge, the fastest best-case signal transition meets the library hold-time requirement and the slowest worst-case transition meets the required setup-time for the next clock edge. The *Combined* signal in Figure 6 represents an amalgamation of the RTL, best-case (BC) and worst-case (WC) signal timing and is shown to illustrate the format used for the Vera input signal in subsequent timing diagrams.

Since the signal transition time varies for the different conditions the *async* modifier can cause undesirable effects in gate-level simulations. If the synchronization to the asynchronous data transition is followed by asynchronous drives or samples the testbench has lost control of the timing events in the gate-level simulation and numerous hazards will result including timing violations and tester time-set matching issues. Figure 7 illustrates the variable timing of Vera outputs if asynchronous drives are combined with asynchronous synchronization as shown in the code snippet:

@(posedge port.$input async);
port.$output = value async;

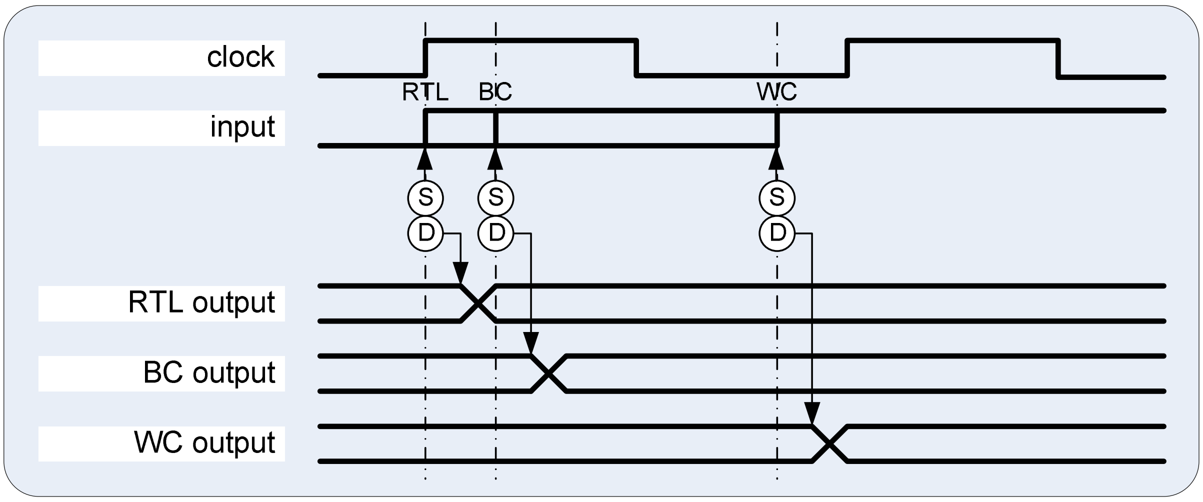

Figure 7: Async Data Transition with Async Drive

The (S) symbols in Figure 7 represent synchronization to the input signal transitions and the (D) symbols represent the subsequent output drives. Since the simulation time does not advance in order to perform the asynchronous drive, the (S) and (D) are aligned in time and shown as touching in the diagram. Driving signals is discussed more fully in Section 4.3 but for the moment note that the asynchronous drive is still subject to output skew resulting in a delay before the signal makes a transition as shown in Figure 7.

If the asynchronous signal transition is followed by a synchronous drive (or sample) relative to the same interface clock edge the *async* modifier has no effect on the resultant timing as shown in Figure 8 and Figure 9.

@(posedge port.$input async);
port.$output = value;

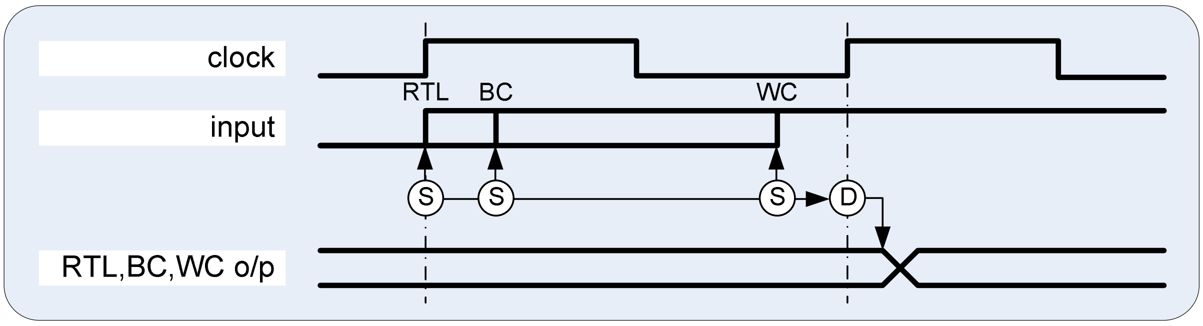

Figure 8: Async Data Transition with PHOLD Drive

@(posedge port.$input);
port.$output = value;

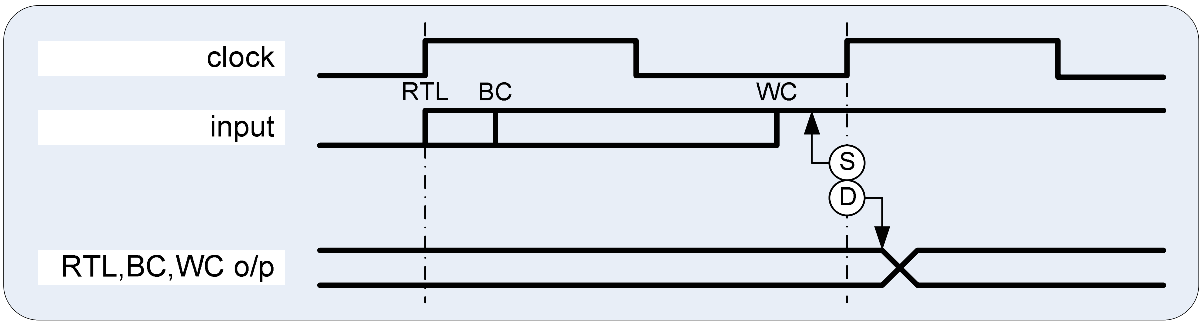

Figure 9: Sync Data Transition with PHOLD Drive

In Figure 8, after asynchronously detecting the corresponding input signal transition, (S), the simulation time advances to the next positive edge in order to perform the synchronous *PHOLD* drive, (D). The same *PHOLD* edge is used for all simulation conditions because the input signal transitions all occur within one clock period. In Figure 9 the input signal transition is not synchronously detected until (S) for all conditions, which is aligned with a rising edge of the interface clock and so the *PHOLD* drive, (D), can occur immediately with no need to advance to the next clock edge.

Note however that if the asynchronous signal transition is followed by a synchronous drive (or sample) relative to a different interface clock edge the *async* modifier results in unreliable operation and potential differences between worst-case and the other simulation conditions as shown in Figure 10.

@(posedge port.$input async);
port.$output = value;

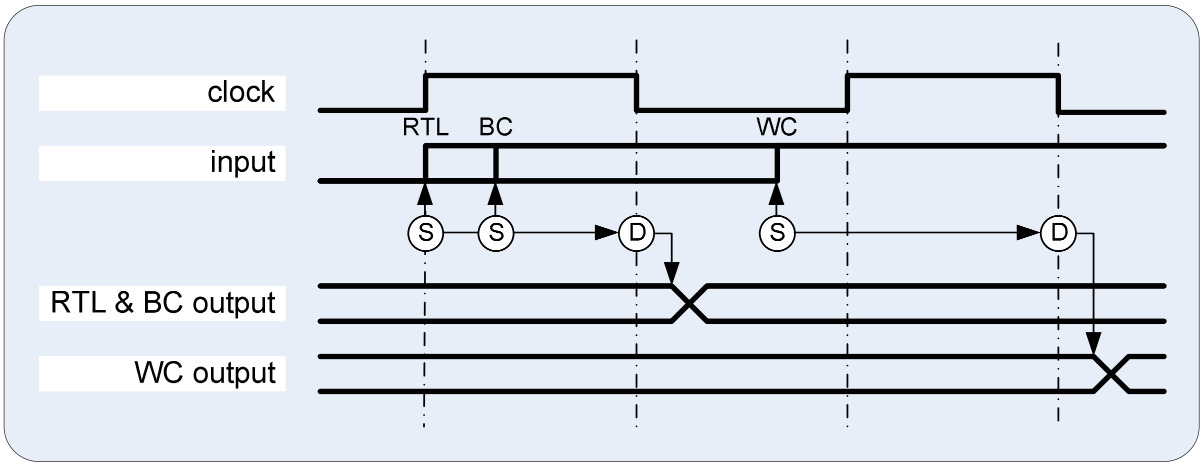

Figure 10: Async Data Transition with NHOLD Drive

In this case Figure 10 shows that the synchronous drive, (D), happens at a different falling edge of the interface clock following the asynchronous transition detection, (S), for worst-case compared to best-case and RTL. In Figure 11 however the drive timing is the same for all simulation conditions since the input signal transition is not synchronously detected until the rising edge at (S) and the simulation advances to the falling edge at (D) in order to perform the synchronous drive.

```
@(posedge port.$input);
port.$output = value;
```

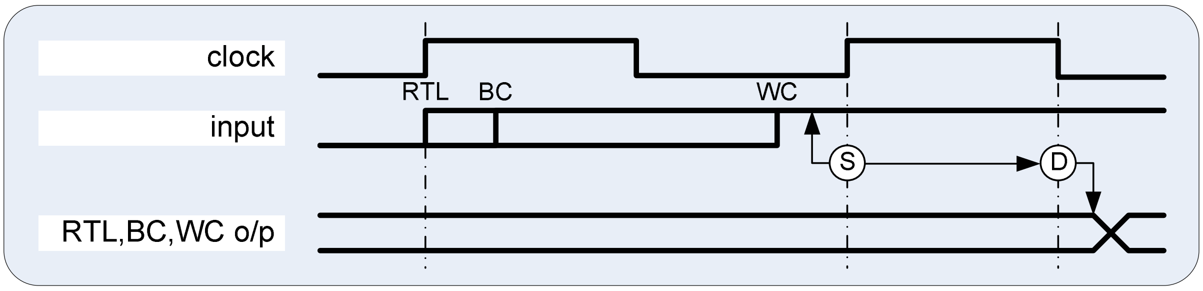

Figure 11: Sync Data Transition with NHOLD Drive

The *async* modifier has no effect on timing when explicitly synchronizing to a clock signal since the virtual port clock should be bound to an interface clock [A5], which by definition can have no skew. The following lines of code produce identical results under these circumstances:

```
@(posedge port.$clock async);
@(posedge port.$clock);
```

In conclusion it is recommended that you do not use the *async* modifier to synchronize to either a data or a clock signal [A6]. If asynchronous drives or samples are present or there are interfaces which use a different clock edge then the asynchronous signal transition could lead to unreliable gate-level operation. If these hazards do not exist then the *async* modifier has no effect on the simulation timing and is therefore misleading.

Since the embedded interface clock may not be closely related to the default *SystemClock* in the SoC testbench environment (e.g. the module bus is divided-by-two) it is recommended to never synchronize to the default clock [A7]. The following code shows examples of potential problems with synchronizing the default clock:

```
// wait for 1 possibly inappropriate SystemClock then drive value
@(posedge CLOCK);                  port.$output = value;

// wait for 5 possibly inappropriate SystemClocks then sample value
repeat(5) @(posedge CLOCK);        var = port.$input;
```

If the *virtual port* contains a clock signal then explicit synchronization should be used (e.g. *@(posedge port.$clock);*) otherwise implicit synchronization should be used. If the event that follows the synchronization is a blocking drive then the *@delay* construct can be used. If however the next event is a sample then a *void* expect can be used with @*delay* to provide the necessary effect. Consider the following examples which show alternatives to synchronizing to default clock:

```
// wait for 1 corresponding interface clock then drive value
@1 port.$output = value;

// wait for 5 corresponding interface clocks (expect nothing!) then sample value
@5 port.$input == void;                var = port.$input;
```

This coding seems long-winded where the next action is a sample, but the *SystemClock* could be a completely different period and shape to the corresponding interface clock due to dividers or PLL clock generators. If interface clock cycle delays are an important part of the protocol then the port definition should contain a clock signal [A8] and explicit synchronization can be used; otherwise the delay is probably not required and should be removed.

The use of the *delay()* system task can cause synchronization issues since it blocks Vera execution until a specified amount of time elapses [A9]. Additionally since the SoC timescale and clock period may be different to the module-level unreliable operation can occur; consider for example a loop that executes 32 times and contain a *delay(1)* call, this would take a different number of fractional clock periods in the SoC environment with a 24ns clock compared to the module RTL environment with a 100ns clock.

Although not a synchronization issue it is worth pointing out at this stage that the *get_cycle()* system function is also susceptible to unreliable operation in the SoC environment if it is used with the default clock [A10]. In this case the solution is to pass in an appropriate signal reference to the function to enable it to reference the cycle count for the corresponding interface clock, for example *get_cycle(port.$input)*.

4.3 Driving Signals

Heuristically it is a bad idea to drive asynchronously in gate-level and tester-compliant simulation environments for the following reasons:

- Uncontrolled drive times may violate gate-level timing resulting in simulation errors
- Irregular signal transition times may violate tester time-set matching and cause pattern processing problems

Note that skew values specified for output signals in the interface definition also apply to *async* drives [3]. If the *async* drive occurs immediately after an *async* synchronization event, as shown previously in Figure 7, then the resultant timing will violate gate-level and tester requirements. If however the *async* drive occurs when the Vera is explicitly or implicitly synchronized to the appropriate clock edge then the *async* modifier has no effect as shown in Figure 12; in this case all the code examples give the same effective timing and yet they imply different intent.

```
// explicit sync to port clock then async drive
@(posedge port.$clock);        port.$output = value async;

// explicit sync to port clock then PHOLD drive
@(posedge port.$clock);        port.$output = value;

// implicit sync to PSAMPLE then async drive
value = port.$input;           port.$output = value async;

// implicit sync to PSAMPLE then PHOLD drive
value = port.$input;           port.$output = value;
```

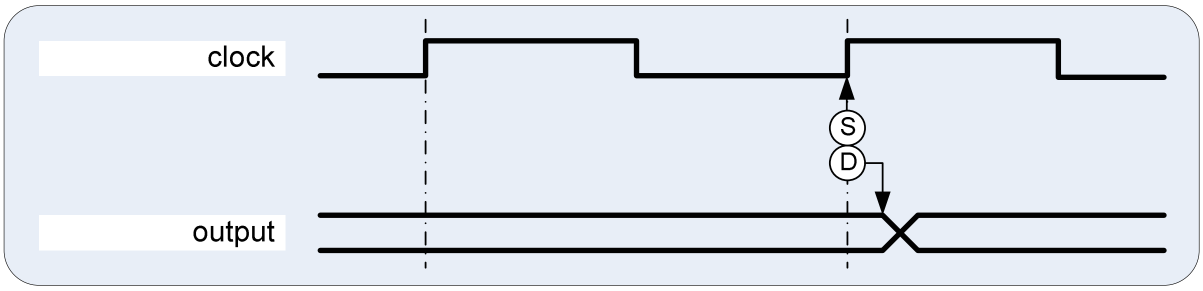

Figure 12: Async and PHOLD Drive After Posedge

If the Vera happens to be aligned with the alternate clock edge to the drive value specified in the interface definition then the *async* modifier affects the timing of the drive as shown in Figure 13. In this case the *async* drive uses the output skew value from the interface definition but effectively ignores the specified *PHOLD* edge (which is inconsistent and misleading) whereas the synchronous drive timing is fully captured by the interface definition.

```
// explicit sync to port clock then async drive
@(negedge port.$clock);        port.$output = value async;

// explicit sync to port clock then PHOLD drive
@(negedge port.$clock);        port.$output = value;

// implicit sync to NSAMPLE then async drive
value = port.$input;           port.$output = value async;

// implicit sync to NSAMPLE then PHOLD drive
value = port.$input;           port.$output = value;
```

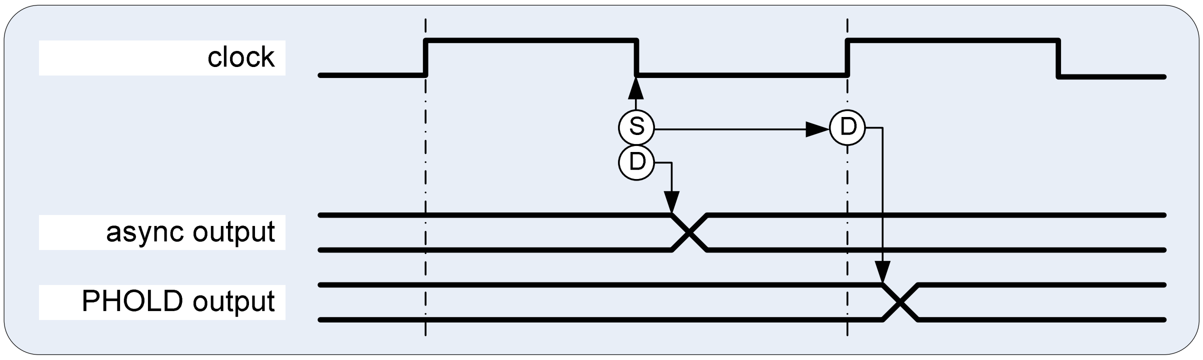

Figure 13: Async and PHOLD Drive After Negedge

Since the affect of *async* drives depends so much on the previous code and the current alignment of Vera we would recommend *async* drives are not used [A11]. Note that it is possible to get an asynchronous drive to happen exactly when the code is executed by specifying an interface output skew of zero but this is not recommended [A12] since the setup and hold timing could be violated for gate-level environments.

It is not recommended to use opposite edges of the clock to control timing in gate-level simulations unless it is explicitly required by the corresponding interface protocol [A13] and the module has been synthesized accordingly. Synthesis tends to maximize worst-case path delays to just fit the available time budget and hence worst-case timing may give different results from best-case and RTL as shown in Figure 14:

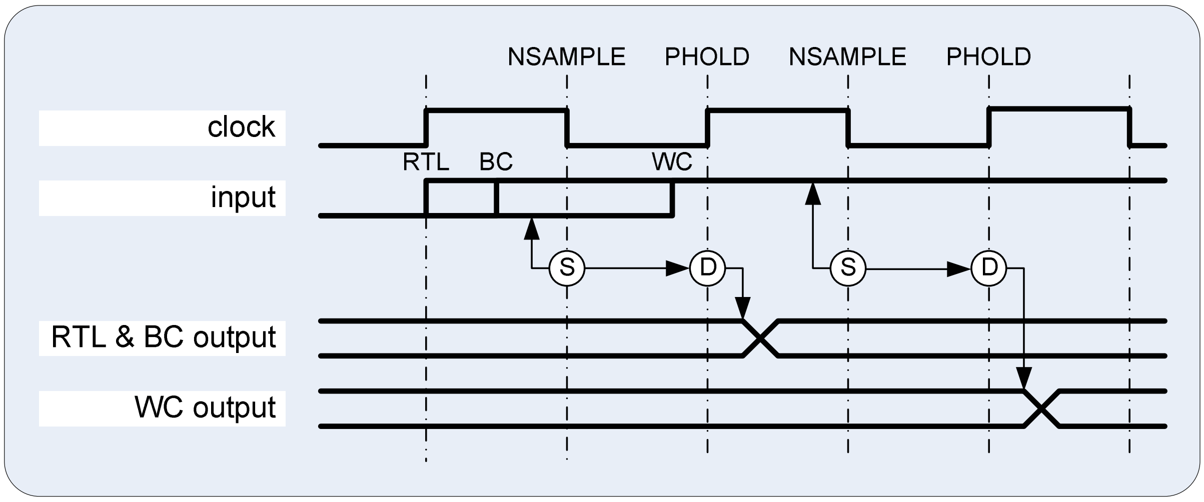

Figure 14: Using Opposite Edges to Control Timing

In Figure 14 the module has been synthesized for the full clock period but the Vera interface is attempting to sample the input using the *NSAMPLE* edge, (S). In this case the subsequent output drive, (D), is different for worst-case compared to best-case and RTL conditions. Note that if a *PSAMPLE* had been used with an *NHOLD* then the input path from the Vera drive to internal registers would exhibit a similar timing hazard resulting in potential timing violations in the worst-case gate-level simulations.

If the interface protocol defines different sampling and driving edges then these should be used for corresponding Vera interfaces in order to fully validate gate-level timing since the clock signal mark-to-space ratio may get distorted due to different rise and fall times. Figure 15 shows the relative timing in this case. Note that the worst-case delay for the input signal is less than half a clock period and should account of potential clock distortion.

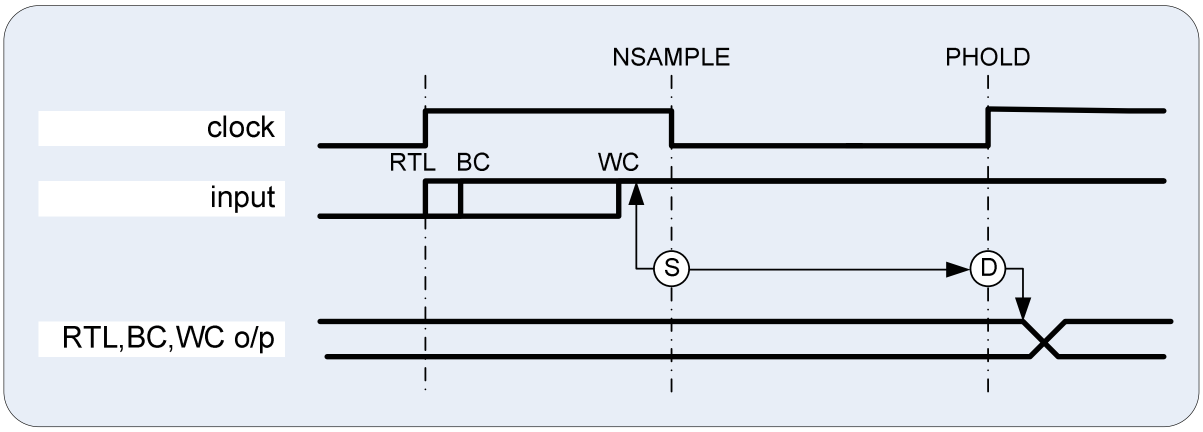

Figure 15: Opposite Edges Required by Protocol

4.4 Sampling Signals

Vera verification IP that is intended to be used to monitor internal signals in a gate-level environment must take into account the effects of clock-skew. We have already demonstrated that the interface clock must be connected to an appropriate clock in order to sample internal signals but in a gate-level environment the clock-tree will have a finite amount of clock skew, hence different terminal points in the same clock-tree have different timing. If the device implementation ensures hold-times greater than the maximum clock-skew then interfacing is relatively straightforward, however many libraries support negative hold-times to facilitate operation with increased clock-skew and ease clock-tree synthesis. In an implementation that supports negative hold times the data into a flip-flop is allowed to change prior to the clock edge without timing violations occurring. Figure 16 illustrates the timing scenario which typically manifests itself under best-case conditions:

```
@(posedge port.$clock);
var = port.$bus async;          // async sample
var = port.$bus;                // sync sample
```

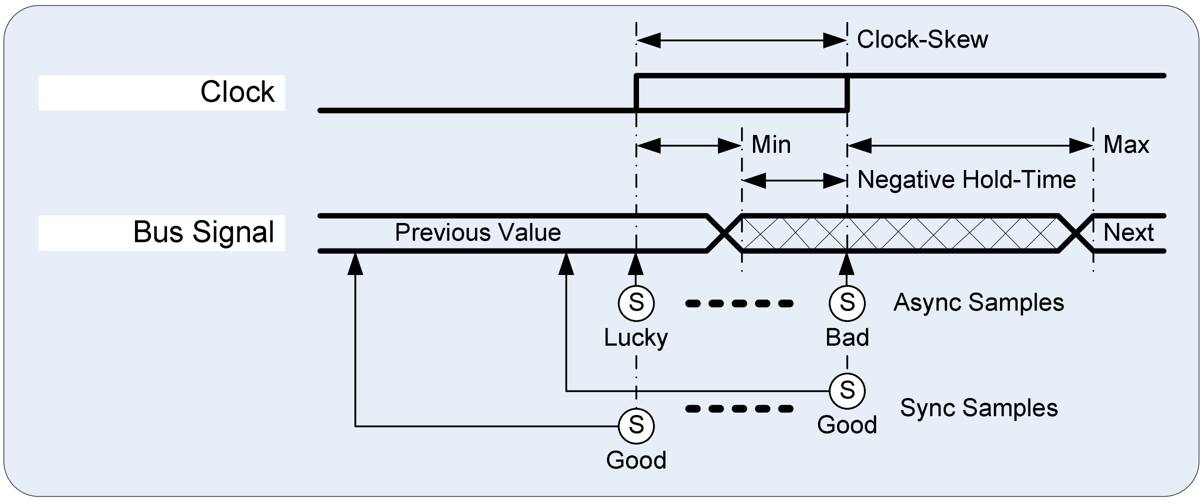

Figure 16: Clock Skew with Negative Hold Times

In Figure 16 the Vera code is explicitly synchronized to an appropriate interface clock which is connected to one of the endpoints of the corresponding clock-tree. The rising edge of the interface clock can occur anywhere in the clock-skew range shown. The negative-hold time capability means that the fastest bit (from an early clock edge, shown as Min in Figure 16) is allowed to change to the new value before the latest clock edge occurs. If we happen to connect and synchronize to a clock net that has maximum skew then asynchronous samples will detect a mixture of new and old bit values and the sampled bus value will be wrong (shown as Bad (S) in Figure 16). If we happen to connect and synchronize to an early clock edge then asynchronous samples might work with a particular release of the netlist but may not work in the next release (shown as Lucky (S) in Figure 16). If synchronous sampling is used with an appropriate negative skew then the bus signal value is always sampled correctly irrespective of the particular clock edge we connect to (shown as Good (S) in Figure 16).

For Vera code to operate reliably under these conditions it should never sample signals asynchronously, even when you are synchronized to the appropriate interface clock edge. Specifically signals should not be sampled using the *async* modifier [A14] and they should not be sampled in expressions [A15]. If it is a single-bit signal that is being sampled the result would typically be mismatches between best-case and other simulation conditions, however if a multi-bit signal is being sampled, e.g. a bus value, then the data integrity could be corrupted if some bits transition faster than others. It should also be clear that asynchronous sampling will also not work if the Vera fails to synchronize to the appropriate clock or sub-cycle delays are used. Some examples of asynchronous sampling are shown below:

```
var = port.$input async;        // using async modifier
if (port.$input == 1) {}        // sampling in an expression
var = port.$input + 1;          // sampling in an expression
```

The recommended solution is to always sample synchronously relative to the appropriate interface clock signal using an input skew value which is greater than the maximum clock-skew. Note that the sampling code is embedded in the verification IP and is independent of the actual value selected for the input skew which is SoC implementation dependant. Here are some alternative implementations that avoid sampling in expressions:

```
 var = port.$input;          // synchronous sample
 if (var == 1) {}            // then use variable in expression

var = port.$input;           // synchronous sample
var = var + 1;               // then use variable in expression
```

In most cases sampling of external pins could in fact be performed asynchronously if explicit clock synchronization is used, however this would limit the reuse potential of the Vera verification IP since it could not be applied to the same interface embedded in an alternate SoC.

5.0 Conclusions and Recommendations

By analyzing the generic issues involved when interacting with a gate-level SoC in a tester-compliant manner we have derived a number of pragmatic guidelines for Vera code development to ensure robust operation across a wide range of simulation conditions from module-level RTL through to tester-compliant gate-level SoC environments. The guidelines can be applied to verification IP development, used during code reviews to assess reusability and as a reference when debugging system-level simulation failures and reliability issues.

6.0 Acknowledgements

We would like to thank the following people for their contributions in reviewing the draft paper material and their general encouragement: Jason Sprott and Davie Robinson of Verilab Ltd., Norbert Huemmer of Motorola GmbH., and Irina Sturm of ST Microelectronics.

7.0 References

[1] ESNUG 421 #1, http://www.deepchip.com/items/0421-01.html , Ed. John Cooley.
[2] Design-for-Test for Digital IC's and Embedded Core Systems, Alfred L. Crouch.
[3] Vera User's Manual, Synopsys.
[4] The Art of Verification with Vera, F. Haque et al.

8.0Appendix: Vera Guidelines

This appendix provides a summary of the guidelines derived and discussed in the paper. These guidelines should be applied when Vera code is intended to be used or reused in tester-compliant gate-level verification environments as well as the SoC and module RTL simulations. These guidelines are just that – guidelines; the reader is encouraged to consider applicability and alternatives in their specific circumstances when using this paper as a reference.

Index to Guidelines:

1. Connect interface clock to an appropriate signal
An interface that is defined to monitor (or drive) internal signals should use an appropriate internal clock. It is not appropriate to specify or default to using the testbench SystemClock since the timing will be wrong for gate-level and the internal interface may be in a different clock domain. **Particular care is required to identify appropriate clocks in the presence of synthesized clock trees and low-power modes of operation. If appropriate, external interfaces should explicitly connect to SystemClock for clarity.**

```
interface internal_if {
  input clk  CLOCK       hdl_node "testbench.top.module.clk";
  input data PSAMPLE #-1 hdl_node "testbench.top.module.data";
}
interface external_if {
  input clk  CLOCK       hdl_node "testbench.SystemClock";
  input data PSAMPLE #-1 hdl_node "testbench.top.data";
}
```

2. Always specify a clock in the interface definition

The Vera interface definition is a mechanism for grouping signals into clock domains and therefore the clock should always be specified. If a clock is not specified the default SystemClock will be used and this might not be appropriate especially in an SoC environment.

Even if a clock does not exist in corresponding ports that are bound to the interface (since the clock does not form part of the port protocol) a clock should be defined and connected in the interface.

For tester-compliant simulations there is no such thing as a fully asynchronous interface since all interaction with the DUT will be relative to the tester cycle which is (generally) matched to SystemClock.

```
// The following port bind does not care about a clock since it
// does not form part of the protocol.
bind interrupt_port interrupt_bind {
  interrupt interrupt_if.interrupt;
}
// However the interface definition should explicitly specify and connect
// to an appropriate clock so that default SystemClock is avoided.
interface interrupt_if {
  input clk       CLOCK       hdl_node "testbench.top.module.clk";
  input interrupt PSAMPLE #-1 hdl_node "testbench.top.module.int";
}
```

3. Do not mix internal and external signals in the same interface definition

Internal and external signal interfaces have different clock requirements for gate-level operation and hence they cannot share the same interface definition.

Since it is not always possible to accurately predict which signals will be internal and which will be external when a module is reused in an SoC the interface definitions are the responsibility of the SoC integrator. Where possible this should be anticipated by the designer of the Verification IP.

```
// Refactor the following mixed interface for module-level VIP...
interface mix_if {
  input clk  CLOCK       hdl_node "testbench.top.clk";
  input bus  PSAMPLE #-1 hdl_node "testbench.top.bus";
  input pin  PSAMPLE #-1 hdl_node "testbench.top.pin";
}
// ...into separate interfaces for SoC operation
interface internal_if {
  input clk  CLOCK       hdl_node "testbench.top.module.clk";
  input bus  PSAMPLE #-1 hdl_node "testbench.top.module.bus";
}
interface external_if {
  input clk  CLOCK       hdl_node "testbench.top.clk";
  input pin  PSAMPLE #-1 hdl_node "testbench.top.pin";
}
// In this case the port definition and all other VIP files are
// unchanged but the bind is modified accordingly.
bind mix_port mix_bind {
  bus internal_if.bus;
  pin external_if.pin;
}
```

4. Connect to module ports, not nets, in interface definitions

Hierarchical synthesis techniques typically employed in SoC methodologies normally

preserve module port names but do not guarantee preservation of net names internal to the modules. Accordingly module-level verification IP that is intended to be reused in a gate-level SoC environment should interact only with module ports and not internal nets.

If an important grey-box net name must be monitored and it is not a module port then the synthesis tool should be instructed to preserve the net using a *dont_touch* attribute or similar – a better solution is to refactor the verification IP to operate with a different signal in the hierarchy.

```
–
```

5. Bind port clocks to interface clocks

Where a clock is included in a port definition it should be bound to an appropriate interface clock and not an interface data signal with zero skew. Ports can have multiple clocks (if required) but they should each be bound to different interfaces.

```
// Replace this interface definition...
interface dual_if {
  input fast_clk CLOCK      hdl_node "testbench.top.fast_clk";
  input slow_clk PSAMPLE #0 hdl_node "testbench.top.slow_clk";
}
// ...with two proper separate interface definitions, and...
interface fast_if {
  input clk CLOCK hdl_node "testbench.top.fast_clk";
}
interface slow_if {
  input clk CLOCK hdl_node "testbench.top.slow_clk";
}
// ...bind the port to both interfaces.
bind dual_port dual_bind {
  fast_clk fast_if.clk;
  slow_clk slow_if.clk;
}
```

6. Do not synchronize using async

Do not synchronise to a signal using async since the transition time will be different between RTL and gate-level simulations. If other asynchronous code exists (e.g. accidentally sampling a signal in an expression) then gate-level simulations will be unreliable.

If the signal is bound to an interface clock then there is implicitly zero skew and the async modifier has no effect. In these circumstance the async modifier should be removed to simplify lint operations and code reviews.

```
// Replace...
  @(posedge port.$sig async);
// ...with
  @(posedge port.$sig);
```

7. Do not synchronize to default clock

Do not synchronize to the default SystemClock since the clock may not have the appropriate timing for gate-level and internal SoC interface signals.

If the virtual port has a corresponding clock signal then use explicit synchronization, otherwise implicit synchronization should be used.

```
// The following code waits for 5 possibly inappropriate SystemClocks
  repeat(5) @(posedge CLOCK);

// If a port clock exists (and it should if we care about protocol
// delays!) then explicit synchronisation can be used
```

```
  repeat(5) @(posedge port.$clk);

// Otherwise implicit synchronisation to the corresponding interface
// clock can be used. If the next action is a drive @delay can be used:
  @5 port.$output = value;

// If the next action is a sample @delay cannot be used, but a
// void assert can be used to block simulation for @delay time:
  @5 port.$input == void;
  var = port.$input;
```

8. Specify clock in port definition if it is part of protocol

If the protocol associated with a port requires explicit operation with respect to a clock (for example a delay of N clocks) then the port definition should include a clock. This enables explicit synchronization to be used in the associated verification IP which provides clear intent and reliable operation if the port clock is bound to an appropriately connected interface clock.

The corollary of this is that where a protocol does not require explicit clock based operations then no port clock should be specified and the associated verification IP should use implicit synchronization and should not include any clock protocol operations like @delay or repeat().

```
// In the module-level testbench the bus monitor used default SystemClock
port bus_port {addr;}
repeat(2) @(posedge CLOCK);
address = port.$addr;

// Instead it should have explicitly defined, connected and used
// a clock appropriate to the protocol:
port bus_port {clk; addr;}
repeat(2) @(posedge port.clk);
address = port.$addr;
```

9. Avoid using the delay() system task

The delay() system task blocks Vera execution until a specified amount of time elapses, using the current timescale. Since verification IP may be reused in SoC testbenches of unknown timescale and frequency unreliable operation can occur.

If the delay is used inside a for-loop the total delay will be an indeterminate number of SoC clocks cycles when the loop is executed. If everything else in the testbench is synchronous then delays will be quantized to the next clock edge. If other asynchronous constructs are present in the testbench then unreliable operation and timing violations may result during gate-level simulations.

```
-
```

10. Do not use get_cycle() system function with default clock

Since the default SystemClock may be inappropriate for verification IP operating on an embedded interface unreliable operation can result from using get_cycle() with the default clock. Instead pass an appropriate virtual port signal in the function call.

```
// Get the number of cycles of SystemClock:
get_cycle();
// Get the number of cycles of my clock:
get_cycle(port.$signal);
```

11. Do not drive using async

If the verification IP contains any asynchronous timing events then driving using async will cause timing issues for gate-level operation and time-set problems for tester-compliant simulations. If the verification IP is correctly synchronized to the corresponding interface clock (either explicitly or implicitly) then the async modifier will not affect the timing, since the drive skew is still used, and therefore the intent is lost and the code is misleading.

```
// Replace...
  port.$output = value async;
// With
  port.$output = value;
```

12. Do not specify zero skew in interface definitions

A drive skew of #0 may cause hold-time violations in gate-level simulations. It is better to ensure that the drive skew covers the minimum hold-time requirement for all simulation conditions.

A sample skew of #0 may cause data integrity problems in gate-level simulations especially if the implementation library supports negative hold times. Sample skew values for internal interfaces should be set to the required setup-time for the implementation library and may need to take into account clock skew and negative hold-times. For external interfaces the sample skew should be set to the guaranteed setup-time for external interfaces.

```
// Use minimum skew values of +/- 1 for drive and sample respectively.
interface example_interface {
  input  clk      CLOCK;
  input  dut_out  PSAMPLE #-1;
  output dut_in   PHOLD   #1;
}
```

13. Do not use opposite edges to control timing unless required by protocol

Using opposite edges of a clock to control the timing is not appropriate for gate-level simulations unless it is explicitly required by the protocol. Maximum path delays will inevitably be more than half the clock period (unless synthesized to the multi-edge clock protocol) and hence worst-case operation will be different from best-case and RTL.

The corollary is that appropriate edges should be used if explicitly required by the protocol in order to validate operation in the presence of distorted clock signals or where the protocol allows a clock to stop in its inactive state. Where a protocol requires both edges to be used care is required to avoid accidentally advancing simulation time due to implicit synchronisation to alternate edges.

```
-
```

14. Do not sample using async

Asynchronous sampling of signals can lead to differences between RTL and gate-level simulations and bus signal integrity issues.

```
// Replace
  var = port.$sig async;
// With
  var = port.$sig;
```

15. Do not sample using expressions

Expressions result in asynchronous sampling of signals which can lead to differences between RTL and gate-level simulations and bus signal integrity issues.

The implicit asynchronous sampling that occurs as a result of using an expression is harder to detect that explicit asynchronous sampling as shown in the following examples.

```
// The following are examples of asynchronous sampling in expressions:
  if (port.$sig == 1) {}
  var = port.$sig + 1;

// Instead perform a synchronous sample then use the variable in the
// expression:
  var = port.$sig; if (var == 1) {}
  var = port.$sig; var = var + 1;
```