2020 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Make your Testbenches Run Like Clockwork!

Markus Brosch, Senior Staff Engineer Digital Verification, Infineon Technologies Austria AG, Villach, Austria (*markus.brosch@infineon.com*)

Salman Tanvir, Staff Engineer Digital Verification, Infineon Technologies AG, Duisburg, Germany (*salman.tanvir@infineon.com*)

Martin Ruhwandl, Senior Principal Functional Verification, Infineon Technologies AG, Munich, Germany (*martin.ruhwandl@infineon.com*)

***Abstract*— This paper outlines a SystemVerilog/Specman UVC architecture that enables efficient driving/monitoring of all clocks within a testbench. Handling of such signals can be a complex and time-consuming task, especially when dealing with IP's that rely on several clocks that may or may not be related to each other. Hence a generic and reusable verification IP can provide many advantages. We first discuss the various features that such a UVC should offer. These include configuration of timing characteristics e.g. period, duty cycle and jitter, handling of asynchronous and synchronous clocks and synchronous reconfiguration. Additionally, permanent monitoring of the clock frequency and checks e.g. whether the clocks are running as expected, are useful. We then describe a prototype UVC and show how the aforementioned features can be implemented. Finally, we conclude with the results of this work.**

Keywords*— *Clock Generation; UVM; SystemVerilog; Specman; DPI-C; Reuse;

I. INTRODUCTION

Handling of clocks is an integral part of verifying any synchronous IP. This includes driving the clock related inputs e.g. clocks and clock gate enables and also monitoring these inputs and other internal derived clocks. For standardized/proprietary protocols, mature reusable verification IP (VIP) is usually available. Using such time-tested IP can offer various advantages such as proven quality, faster verification environment ramp up and ease of use which culminates in a shorter turnaround time. However, for clock related signals, in our experience, we have not found a generic clocks UVC that offers the features that we would consider as vital. In numerous projects we have seen that clocks are handled in an ad hoc manner rather than using a dedicated clocks UVC. This repetitive redevelopment, especially when verifying complex IP's with numerous clocks, wastes precious project resources. Additionally, reusability/portability is compromised. Our goal in this paper is to present a generic reusable clocks UVC that solves these problems. We begin by first defining the feature set that any such UVC should offer. The following chapter then describes the UVC architecture and how it implements the defined features. At the end, we present a conclusion of this work.

II. FEATURES

A. *Reusability*

The Clocks UVC should be generic and able to be reused in different testbenches with a varying number of clocks.

B. *Driving*

The Clocks UVC should be able to generate a reference clock and a varying number of derived clocks mimicking a real clock tree. The reference clock timing characteristics e.g. period, duty cycle and jitter should be configurable. Each derived clock should be configurable in the following way:

- Adjustable phase.
- Low and high time to be specified as a number of edges of the reference clock.
- Fractional clock division.

- Patterns to support pulse sequence generation.
- Duty cycle (based on edges of the reference clock or real time).
- Drive clocks to High Impedance (Z).
- Synchronous reconfiguration without frequency glitches.

Each derived clock should be able to be individually enabled or disabled. Disabling the reference clock should gate all clocks.

C. *Monitoring*

The Monitor should publish the time period and duty cycle of a clock once at the start and then whenever a change is detected. To ensure that a clock has not stopped due to an unexpected reason, the monitor should trigger an error after a configured timeout is reached. If a clock is expected to be disabled, the monitor should also give an error if the clock toggles.

III. ARCHITECTURE

This chapter describes the architectural features of the clocks UVC. These are:

A. *Split transactors*

The VIP is split into two parts: an interface-based HDL part and a class based HVL part as described in [1]. The HDL part consists of BFM's for driving and monitoring whereas the HVL part is the controlling or configuring layer. The HVL part also receives the monitored transaction from the BFM and passes it to subscribing components in the testbench. We opted for this split architecture to be able to have a common HDL part that could be controlled and configured from a SystemVerilog or Specman e HVL part. This has the advantage that only a single set of BFM's needs to be maintained. Although emulation is not a current use case, this split architecture also makes the UVC emulation friendly. If emulation needs to be supported later, the UVC could be adapted with minimal effort.

B. *Testbench to DUV connection*

The BFM's are parameterized to support a variable number of clocks per agent, without being limited by a maximum footprint. For the SystemVerilog HVL to HDL connection we decided to use the abstract class approach. For Specman the SystemVerilog DPI-C interface is used.

1) SystemVerilog abstract class approach

In SystemVerilog, the DUV interface signals are typically accessed via a virtual interface handle placed in the class based HVL environment. This handle can point to a physical interface/BFM in the HDL part, provided they are both of the same type. As each specialization of a parameterized interface is a new type, this mechanism cannot be used for the Clocks UVC. However the abstract class/polymorphic interface approach solves this problem. In this approach, an abstract base class handle is used instead of a virtual interface. The abstract class defines an API to pass information to and from the BFM. A parameterized concrete class that extends from the abstract class implements this API. Hence the abstract class handle can be used to reference any of the parameterized classes via polymorphism. Please refer to [2] for more details on this mechanism.

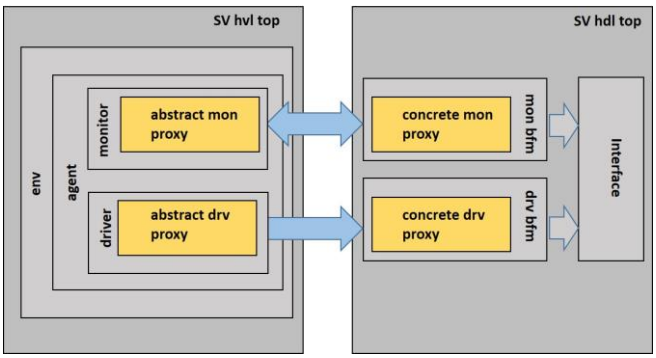

Figure 1. SystemVerilog Abstract Class Approach

2) Specman DPI-C

SystemVerilog supports interfacing with foreign languages via the Direct Programming Interface (DPI-C). This can be used to connect the SystemVerilog HDL BFM with the Specman HVL environment. DPI-C supports the calling of functions implemented in a foreign language from SystemVerilog and conversely, the exporting of SystemVerilog functions to be called in the foreign language. The functions need to be qualified with an "import" or "export" directive depending on the desired direction of communication. The language that implements the function, uses "export" whereas the language from where the function is called, needs to import it. Further implementation details on the Specman/SystemVerilog integration can be found in [3].

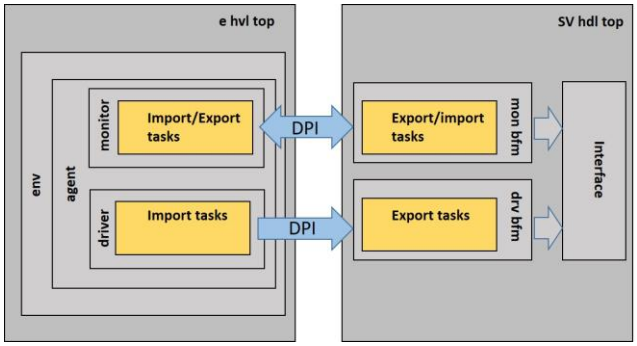

Figure 2. Specman DPI-C

C. *Configuration*

The monitor and driver BFM's are configured via dedicated configuration objects which are configured via the HVL part and passed to the BFM using the testbench to DUV connection as described in section III-B.

The driver BFM's are configured via 2 objects, one for the reference clock and the other for each derived clock. These are illustrated in the following 2 code snippets.

```
class ifx_clocks_driver_ref_clock_config extends uvm_object;
  rand bit [31:0]   clk_high_phase_width;
  rand bit [31:0]   clk_low_phase_width;
  rand bit          clk_enable;
  rand bit          jitter_enable;
  rand int unsigned jitter_factor;  //percentage jitter
  string            clk_name;       //clock identifier
  ...
```

Figure 3. Reference Clock Configuration

Figure 3 illustrates the reference clock configuration. The period/duty cycle can be configured by the clocks low and high phase widths. Additionally control knobs for clock gating and jitter are provided.

```
class ifx_clocks_driver_derived_clock_config extends uvm_object;
  string           clk_name;              //clock identifier
  rand logic       clk_startval;          //initial clock signal level
  rand bit [31:0]  clk_high_phase_width;  //high phase time for mode 1
  rand bit [31:0]  phase_shift;           //phase shift from ref clock
  rand bit         clk_enable;            //clock enable signal ('1'=on / '0'=off)
  rand bit         clk_high_z;            //tristate high z clock out
  rand bit [7:0]   pattern_size;          //pattern size, range = 1 to 128
  rand bit [127:0] enable_pattern;        //modes 0,1: sequence pattern, mode 2: edge counts
  rand bit [1:0]   mode;                  //mode 0 = Sequence patterns
                                          //mode 1 = Sequence patterns with configurable high time
                                          //mode 2 = Edge Counter
  ...
```

Figure 4. Derived Clock Configuration

Figure 4 shows the implementation of the derived clock configuration. The start value of the clock is configurable with the "clk_startval" knob. The "clk_enable" knob is used for clock gating and "clk_high_z" can be used to drive the clock to High Impedance state when gated. All remaining knobs are controlling the different driving modes that are explained in section III-D-2.

```
class ifx_clocks_monitor_clock_config extends uvm_object;
  string   clk_name;    //clock identifier
  rand bit publish_en;  //control knob to enable/disable publishing

  ...
```

Figure 5. Monitor Configuration

Figure 5 shows the monitor configuration object. The "publish_enable" knob provides control over the publishing of the monitor transactions. The need for this is explained in section III-E.

Although the configuration objects are implemented as classes, these cannot be directly passed to the BFM's when using the Specman DPI-C interface or when considering emulation. For these use cases, packed structs are required for communication between the HVL and HDL environments. Hence the underlying communication translates the higher level configuration classes into lower level configuration structs. This also applies to the passing of the monitor transaction from the BFM. The class based layer is not omitted in order to leverage the benefits it provides e.g. randomization and TLM communication.

D. Driving Implementation

The clock generation is implemented in a parameterized HDL BFM. The parameter is used to drive any number of derived clocks from a single generated reference clock. The driver BFM is initialized via the agent configuration which is passed from the HVL environment before the simulation run phase. Subsequent configuration updates can be propagated by a sequence layer which wraps the driver configuration objects and some additional control knobs in the transaction item. The configuration updates in the BFM are implemented using shadow registers which are used in order to align the configuration update with a synchronization point. This can either be the rising edge of the reference clock or when all clock periods align. This synchronous update with the period alignment allows reconfiguring the generated clock frequency without producing any intermediate artifacts.

The generation is implemented in two processes, one for the reference clock and another for the derived clocks.

1) Reference Clock Process

The derived clocks are initialized using the configuration to ensure a defined value before the first reference clock event is captured in the derived clock process. The generation of the reference clock then begins within an endless loop. If the reference clock is disabled, it is gated in the beginning of the low phase. Jitter can be introduced by using the "jitter_en" and "jitter_factor" knobs of the configuration. The "jitter_factor" parameter specifies the percentage variation in the clock period. We opted for a simplistic jitter implementation with the variation only being applied to the high phase of the clock. Figure 6 illustrates an example with a 5% jitter being applied to a 20 ns clock

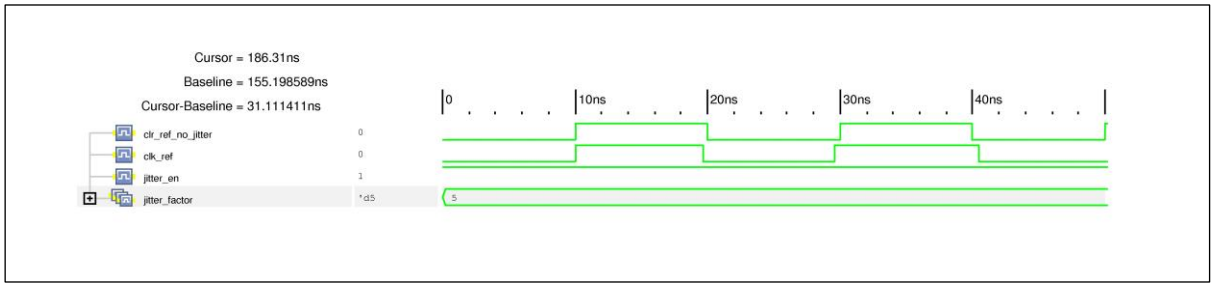

Figure 6. Clock Jitter

2) Derived Clock Process

The derived clock process is triggered on the reference clock. This is a single process implemented within a generate-for-loop construct using the parameterization for the number of derived clocks. Each clock can be configured to use one out of the following three available modes.

In the first mode "Sequence Patterns", the user can control the derived clock generation by specifying a sequence pattern of up to 128 bits using the "enable_pattern" parameter of the configuration

object. The clock generation process is triggered on the positive edge of the reference clock, and incrementally checks each bit of the pattern and propagates the reference clock through where a 1 is specified. The bit pattern size can be set using the "pattern_size" field. The passing through of the reference clock can optionally be shifted by using the "phase_shift" parameter. Figure 7 illustrates an example of a pattern sequence generated clock.

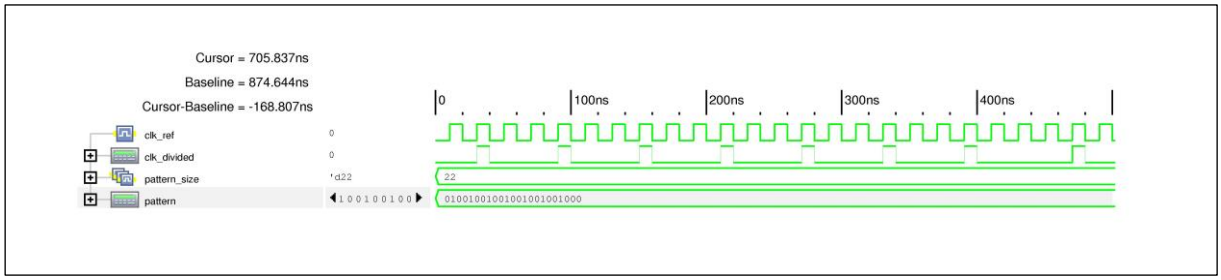

Figure 7. Mode: Sequence Patterns

The second mode "Sequence Patterns with Configurable High Time" is depicted in Figure 8. This works the same way as the first mode with the exception that when a 1 is detected in the pattern, the reference clock is not passed through, but rather the high time of this phase is configurable via the "clk_high_phase_width" parameter of the derived clock configuration.

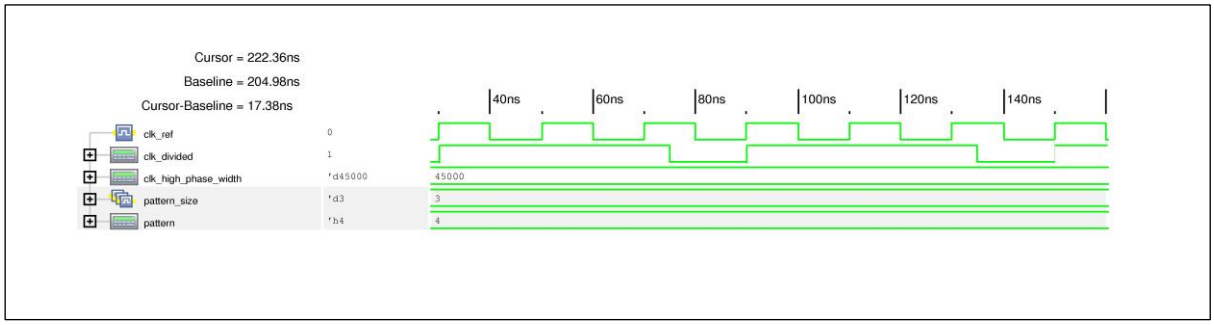

Figure 8. Mode: Sequence Patters with Configurable High Time

The third mode "Edge Counter" allows the user to control the high and low phases by specifying the number of edges of the reference clock in each phase. The "enable_pattern" parameter is reused in this mode to specify the number of edges. The phase shift is also configurable similar to the first two modes.

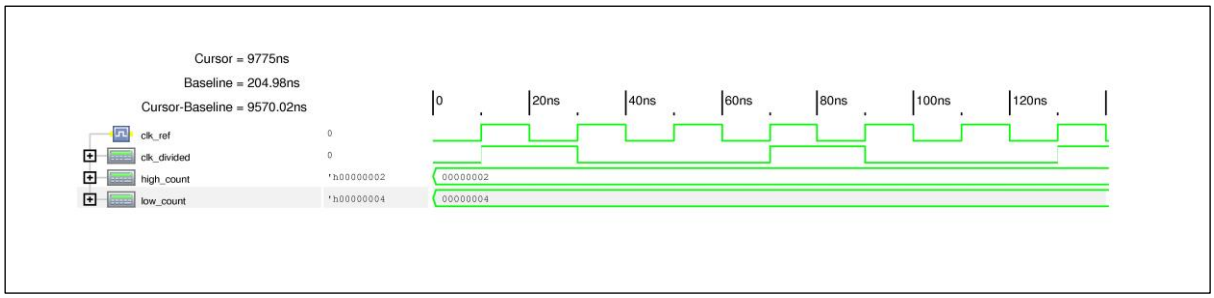

Figure 9. Mode: Edge Counter

As explained in the driver implementation introduction, the active configuration of the BFM can be updated in a synchronized manner. This is illustrated in Figure 10.

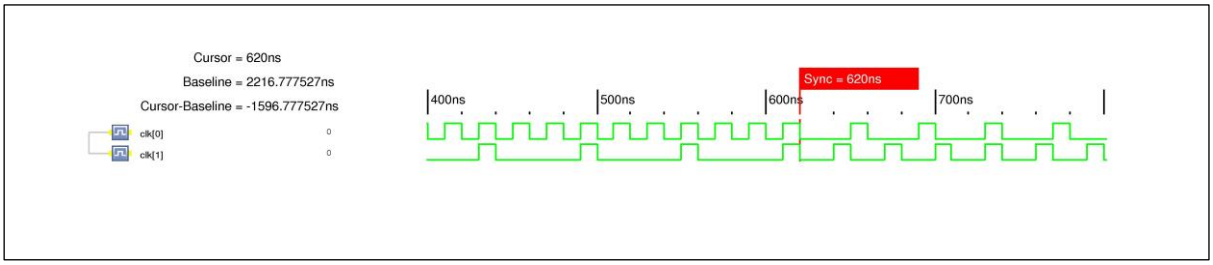

Figure 10. Sync Feature

The pattern sequence modes are implemented to fulfill the requirements to be able to flexibly generate any arbitrary pulse sequence and also be able to achieve fractional clock division. The pattern sequences allow the user to achieve a time averaged fractional division factor. Figure 7 shows how a division factor of 3.14 can be achieved.

E. Monitoring Implementation

The monitoring functionality is implemented in an HDL BFM which can be parameterized to the number of required clocks. It is expected for all clocks that are related to each other to be connected to a single BFM. This facilitates checking of clock relationships within a clock tree. For each group of clocks, a separate BFM can be used.

If configured, a permanent monitoring task is started to measure the time period/duty cycle of each clock. This task is implemented once but replicated using the generate-for-loop construct. The number of threads is derived from the BFM parameter for the number of clocks. Every thread measures the time period and duty cycle for each clock cycle, but only publishes a transaction when a change in either parameter is detected. For clocks that use patterns or clocks with jitter, this publishing scheme could generate a lot of traffic. Hence the publishing can be controlled via the “publish_en” parameter of the configuration.

An alternate task enables the user to measure the period/duty cycle over a configurable sampling period on demand. The task arguments include the number of clock cycles for measurement and additionally, a timeout to guarantee that the task does not get stuck. The task returns an array of monitor items for each clock cycle measurement and an additional item that holds the average measurement over all cycles.

To be able to ensure that the clocks are still running and not stuck, a check can be enabled that raises an error on not detecting any activity within a configurable timeout. Additionally it is possible to check that no toggling is detected on a clock that is expected to be gated.

It is a common use case to be able to synchronize to a clock. To aid this, a task is implemented which makes it possible to synchronize to a configurable number of edges (positive or negative) of the clock.

IV. CONCLUSION

This paper discussed the features that we consider a complete clocks UVC should offer. A prototype was presented to demonstrate how these features can be implemented. We show how a common SystemVerilog BFM can be connected to either a Specman e or a SystemVerilog environment using the abstract class approach or Specman DPI-C interface. The UVC can generate any number of clocks with a configurable set of timing characteristics. These can be dynamically changed with the sync feature without introducing intermediate frequencies. Various monitoring tasks offer the possibility to measure clock timing, synchronize to a configurable no of clock events, and to check that a running clock does not unexpectedly get stuck or that a gated clock stays disabled. Due to the flexibility of the UVC, it can easily be reused in any project.

REFERENCES

[1] Universal Verification Methodology UVM Cookbook, Mentor Graphics Corporation, 2019, pp.372-412

[2] David Rich, “The Missing Link: The Testbench to DUT Connection”, DVCon2013, San Jose, CA.

[3] Specman Integrators Guide, Cadence, 2020, pp.508-564