Run-time Configuration of a Verification Environment - A Novel Use of the OVM/UVM Analysis Pattern

Paul Marriott Verilab Canada
Montreal, Canada
Email: paul.marriott@verilab.com Mark Ronan Diablo Technologies Inc.
Ottawa, Canada
Email: mronan@diablo-technologies.com

***Abstract*—This paper describes a novel approach to modeling the real-time variation of delays required for the functional verification of a DIMM (Dual In-Line Memory Module) system consisting of DDR3 memory and other interface devices using an OVM environments analysis pattern to provide run-time delay control. The system requirements could not be verified without the ability to provide temporal control of the delay elements in the environment. Different approaches to solving this problem were examined before determining the solution adopted was the best fit for the task at hand.**

***Index Terms*—Design patterns; OVM/UVM; ASIC design; timing simulation; SystemVerilog**

I. Introduction

Verification environments typically have several mechanisms for communicating information to their elementary components. Values can be set directly, modified via an API (application program interface) or sent as a transaction through TLM ports and exports. The type of mechanism used is often determined by the frequency with which information has to be communicated. A single value set at the start of simulation might just be forced into an RTL net via a backdoor mechanism; data which is changing at run-time may better be transmitted using a well-defined TLM approach.

There is also a distinction between data that comprises the stimulus to the design (including responses from the DUT) and data used to configure the operation of components. This latter data usually is restricted to the start of a test.

The OVM [1] provides several mechanisms for data communication in a verification environment, as well as the facility to store and retrieve configuration data. Data communication is normally performed through TLM connections (port to export) and is essentially point-to-point.

Verification components can retrieve configuration data from the OVM's configuration database. Indeed, this facility is encouraged and so components are usually responsible for retrieving their own configuration parameters at the start of a test. However, the requirement for the problem at hand was to be able to change configuration settings at run-time as well as to have groups of components all using the same configuration data. An examination of the system under verification will reveal the exact requirements that lead to the solution adopted.

II. The System Under Verification

The board-level system under verification consists of a DDR3 RPLL (Register Phase Locked Loop) device interfaced with memory organized into byte lanes and therefore appears as a DIMM to the host system. A representative DIMM is shown in Fig. 1. The RPLL [4] device re-times and re-drives all the signals required in a DDR3 interface. This allows for the highest operational speed possible. The physical interface (PHY) of the DDR3 memory contains several DLLs (Delay Locked Loops) and PLLs (Phase Locked Loops). The DLLs allow the timing of all of the control signals to be adjusted to each memory chip, allowing for board trace delays etc., in order to meet the DDR3 timing specifications [5].

To achieve the highest system speed possible, the DLLs, which control the launch time of all the memory interface signals, have to be calibrated to the actual board and physical conditions (temperature, voltage and the process corner of the silicon used) present. This calibration is performed automatically by the RPLL device at the time of board manufacturing. Once calibrated, the

internal DLLs compensate for variations in temperature and voltage during the operation of the board.

From a verification completeness perspective, it is essential to be able to model the board delays as well as all the internal DLL loop-control delays to ensure that the automatic calibration can be achieved. Even though the simulation environment is purely digital, all of the sub-cycle timing behavior needs to be verified. The DLL models themselves are essentially black-boxes as far as the verification environment is concerned, but their behavior is such that the sub-cycle timing can be observed and verified. For this latter aspect, assertions were created to check the timing requirements in the DDR3 specification were being met. However, they are beyond the scope of this paper but are covered in [7].

III. Verification Challenges

Verification of this kind of timing is different from the usual gate-level simulations or static timing analysis. The system-level verification environment has delay elements inserted to model both board and trace delays and the internal loop-control delays in the DLLs. Even though the initial verification was performed purely at the RTL level, models were available of the DLLs that meant that it was feasible to use a realistic model of the actual board delays, even without resorting to mixed-signal simulation. Several aspects of the overal board-level environment used behavioural models which accurately accounted for the actual timings of the DDR3 interface signals.

To achieve the verification requirements, several different delay conditions have to be modeled. In reality, temperature and voltage variations mean the DLLs' delay values change over time. To verify the robustness of the DLL control algorithms, a verification environment allowing delay values to be changed on-the-fly during a simulation is required. The challenge was to come up with a mechanism to allow this run-time control but allow some flexibility as well. As the project was progressing, the design of the PHY was being optimized and this meant the usage of its internal delay elements was not necessarily fixed. Since certain groupings of delay elements were necessary for correct operation, the modeling of their timing variation over time (to model the effects of temperature changes, for example), meant that all elements in a particular group would need to have their delay value updated at the same time (and by the same amount).

Figure 2 shows a simplified representation of the board-level delays and a representative PHY showing the internal delays is given in Fig. 3 (from [6]).

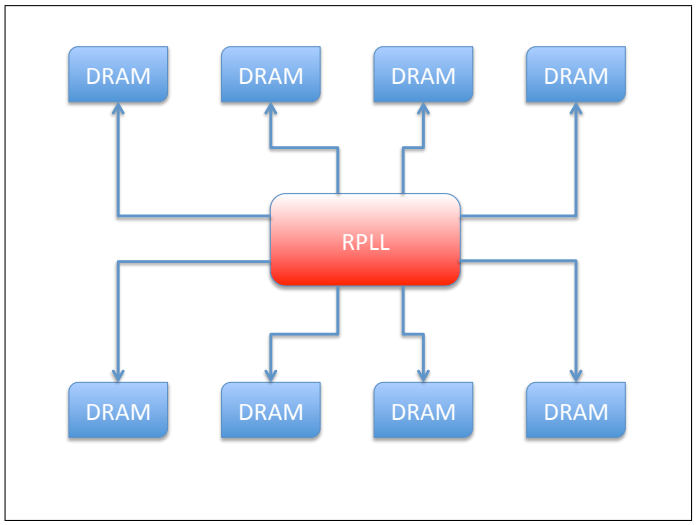

Fig. 1: Typical DIMM

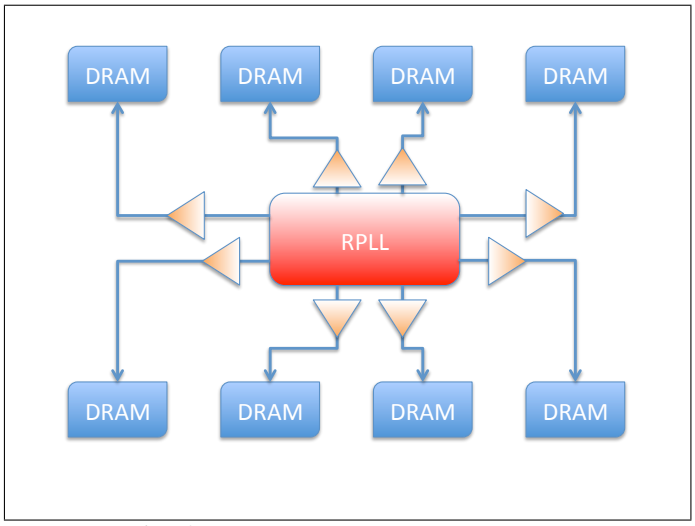

Fig. 2: DIMM showing board delays

IV. Solution Adopted

A. Conventional OVM Approach

In an OVM environment, a typical example of which is shown in Fig. 4, a component can get its own configuration from the internal database via a `get_config()` call.

This approach is fine for components that need to be configured once, but, for various reasons, would be inconvenient and inefficient for the problem at hand. The main reason is due to the run-time variation requirement. To store all the run-time values in the configuration database would be quite onerous, though one could contemplate that all the values could be computed at the beginning of simulation and stored in the database. Another reason is the overhead required to retrieve the configuration values at run-time. As As the database is

essentially an array indexed by string, a string lookup is required each time a value is retrieved from it. String search operations are relatively costly in SystemVerilog and would add an unwanted overhead to the simulation time. As there could be hundreds of delay elements, each time a new value would be retrieved from the database many hundred lookups would be required.

1) Use of Sequences: An investigation was made into using some kind of sequence to generate the required updated values, but this was somewhat limited in that each delay element potentially needed its own set of values again with several hundred delay elements in use, the testbench structure to connect all of these together appeared to be too complex.

2) Drawbacks of Conventional Approach: With an analysis of the architecture of the PHY, it was apparent that many of the delay elements were, in some way, related and would have to be updated at the same points in time during a simulation. These updates would potentially require hundreds of `get_config()` calls which, as well as being inefficient, would make for unnecessary duplication of data as many of the values would, in fact, be the same for many of the elements. Also, the configuration of the association of elements with one another was not completely determined when the verification environment was being created. Having to update both the configuration database as well as the connectivity of the point-to-point control connections was not an attractive proposition.

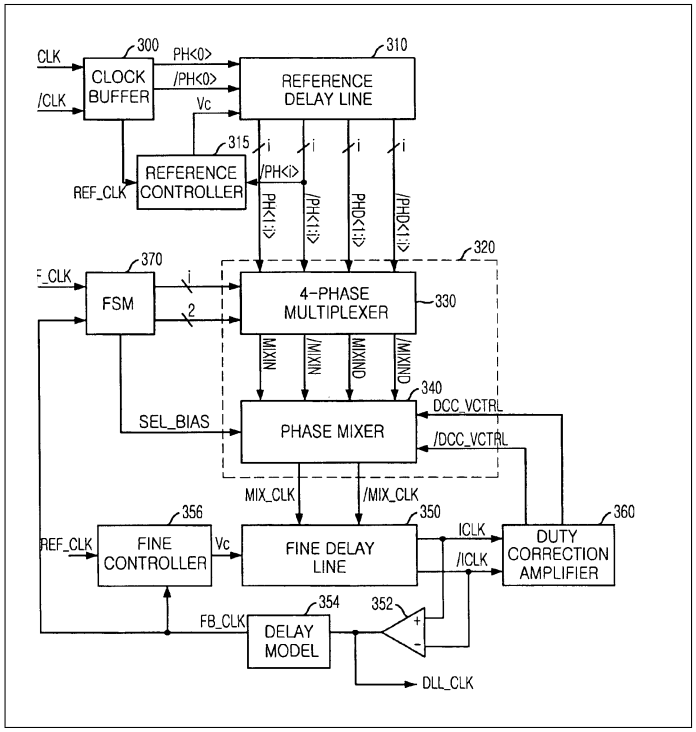

Fig. 3: PHY with delay elements

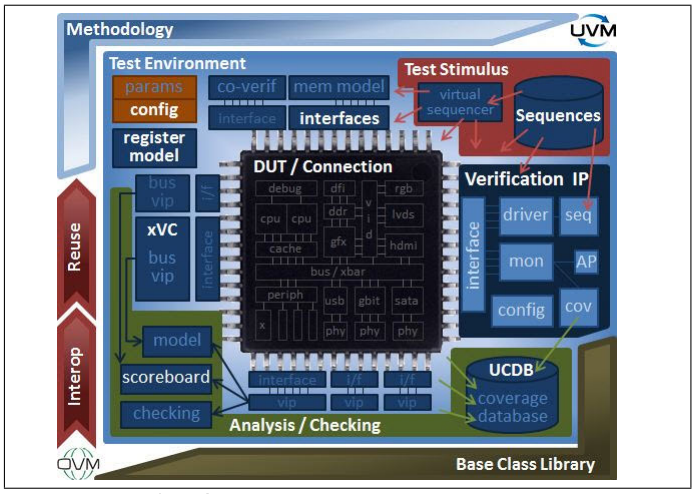

Fig. 4: Typical OVM Environment

B. Adaption of the Analysis Pattern

The "Analysis Pattern" [3] is a software engineering construct used to provide a subscription service, of zero or more subscribers, to a broadcaster. In the OVM and related methodologies, the analysis pattern [2] is normally used to attach scoreboards and coverage objects to transaction monitors, as it provides an implementation of the subscription mechanism that allows multiple interested objects to receive the stream of transactions from a source. As such, it is usually used for data transactions, rather than configuration and control. Typical usages of analysis ports (shown connecting monitors to scoreboards) is shown in Fig. 5 (from Paradigm Works). It was quickly recognized that the analysis pattern would present an ideal solution to the association of delay elements with a particular stream of timing updates. Groups of elements could be created and each group subscribed to the output of a generator that would provide the updated values during a simulation. The subscription mechanism is flexible enough that it was trivial to re-arrange the groupings as the RTL structure of the PHY crystallized.

1) Groups of Subscribers: Allowing a generator to invoke an update method in one or more verification components asynchronously could have been met either by direct method calls or by a TLM blocking port connection. However, the normal TLM connection mechanism is one-to-one and so this would create unnecessary complexity in connecting up the plethora of delay elements, especially as many of them are in the same group. The attraction of using analysis ports is that they provide an especially convenient support for one-to-many and one-to-none connectivity. This latter aspect is important as, during the evolution of the project, certain delay elements were deemed to not vary or be set to

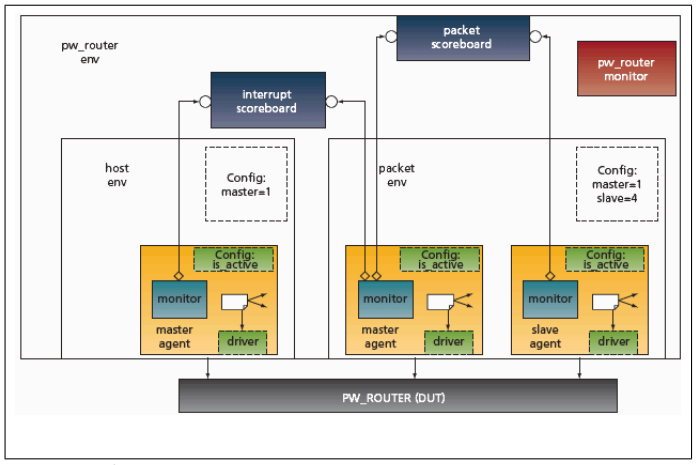

Fig. 5: Typical OVM Analysis Port Usage

zero. With the analysis port approach, it was trivial to remove such elements from a ports list of subscribers. Indeed, certain groups ended up with no subscribers. The analysis pattern is exactly suited to broadcasting common values to groups of subscribers.

2) Configuration Issues: The verification environment itself is configured and built from a randomized configuration object and this was also used to specify the subscriptions of the delay elements into appropriate groups. In the general case, this mechanism provides a way to decouple the connectivity of the testbench from its configurability. Since the analysis pattern allows for zero subscribers to a particular analysis port [2], this gives the maximum possible flexibility as a component can elect not to subscribe at all. Though the control transactions in the environment in question were simple time offsets, the same methodology could easily support a more complex protocol. The groupings of subscribers provide for an arbitrarily complex control orchestration that allows associated components to be configured in concert as may be required.

V. Implementation Details

A. Delay Elements and Control API

The delay elements were implemented as interfaces with a non-blocking signal assignment with the delay value specified as a member of the interface. Both uni- and bi-directional delays were created and techniques employed to ensure the ordering of events, should the delay change with a signal already in-flight.

Listing 1. Simple delay element

```
initial begin
        out = in;
        forever @(in) begin
          out <= #(delay_val) in;
        end
end
```

A concrete implementation of an abstract API class was used to set the `delay_val` member. This mechanism allowed both procedural access to the delay value or update via the analysis port connection.

Listing 2. Abstract API class

```
virtual class delay_api_abstract
        extends ovm_pkg::ovm_subscriber#(time signed);
  time signed default_delay = 0;
  time signed current_delay;
  virtual function void set_default_delay();
     current_delay = default_delay;
  endfunction
  virtual function void set_delay(time signed the_delay)
     current_delay = the_delay;
     // assign the interface delay_val signal
     // with the_delay in the derived class
  endfunction
  virtual function void offset_delay(
                        time signed the_offset);
...
  endfunction
  virtual function void write(time signed t);
  endfunction
...
endclass : delay_api_abstract
```

In the concrete implementation, the actual interface member which is used to store the delay time is assigned and an implementation of the analysis export's `write()` method is created.

Listing 3. Concrete API class

```
class unidir_delay_api
    extends global_type_Pkg::delay_api_abstract;
    virtual function void set_default_delay();
       super.set_default_delay();
       delay_val = default_delay;
    endfunction
...
   virtual function void offset_delay(
                      time signed the_offset);
      super.offset_delay(the_offset);
...
    endfunction
    virtual function void write(time signed t);
       super.write(t);
       this.offset_delay(t);
    endfunction
...
endclass : unidir_delay_api
```

In the interface, an instance of the API class is constructed and a function created to allow the analysis export to be connected, recalling that the API class is derived from ovm_subscriber. Handles to the API classes are wrapper in an interface wrapper class which is then made available to the OVM agent responsible for that particular interface. An initial block in the interface sets the default delay value. Updates can then be made either by calling the appropriate API method, or via the analysis connection. For this project, the ability to vary a

delay from its current value was required so the method used is called offset_delay. Of course, in the general case, any transaction received through the analysis interface could be decoded and further processed.

Listing 4. Partial interface code

```
unidir_delay_api api = new(
    $psprintf("%m.%s","delay-api"), null);

function void connect_ap (ovm_analysis_port
                          #(time signed ) ap);
   ap.connect( api.analysis_export );
endfunction : connect_ap

initial begin
   api.set_default_delay();
end
endinterface
```

The use of `"%m.%s"` in the format string gives the class instance name in the OVM registry the actual path it has in the design hierarchy. This is exploited in the `connect()` methods used to connect the analysis subscribers to their associated broadcast port.

B. Delay Variations

Generators were created that used wavetables in order to create the ability to vary the delay values according to different "shapes" such as ramp, DC triangle (delay increasing and decreasing from a set value), AC triangle (delay increasing and decreasingm centered around a norminal value), sinewave and 32-element random values summed to an average of zero. Fig. 6 shows the visualization of several groups varying with different frequency AC triangle patterns.

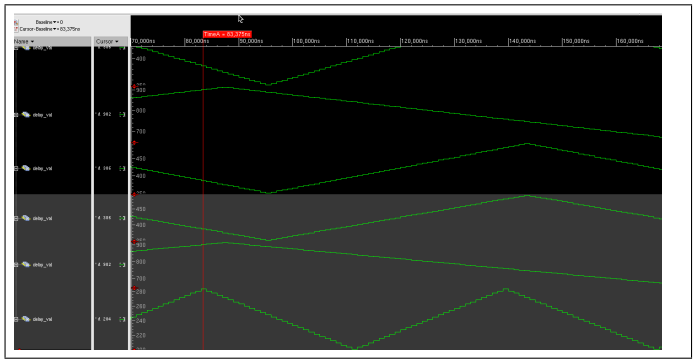

Fig. 6: Varying Delay Values for different groups

The following code snippets show part of the function generator classes for the DC triangle wave and the zero-sum noise wave respectively.

Listing 5. Function generator classes

```
class triangle_wave
      extends delay_function_abstract_lut;
  `ovm_component_utils(triangle_wave)
...
   virtual function void initialize_table
     (int num_elements=128);
     table_elements = num_elements;
     delay_function_table = new[table_elements];
      for (int i=0; i<table_elements/2; i++)
      delay_function_table[i] = 1;
      for (int i=table_elements/2;
           i<table_elements; i++)
      delay_function_table[i] = -1;
   endfunction: initialize_table
...
endclass: triangle_wave
class noise_wave
      extends delay_function_abstract_lut;
  `ovm_component_utils(noise_wave)
...
  rand bit signed [5:0] temp_function_table[32];
  virtual function void initialize_table
    (int num_elements=32);
    assert(randomize(temp_function_table)
     with { temp_function_table.sum() == 0;};
    foreach (temp_function_table[i])
      delay_function_table[i]=temp_function_table[i];
  endfunction: initialize_table
endclass: noise_wave
```

The delay variation generators create an instance of the appropriate function generator class (specified in the overall environment configuration) and then loop through the function table to obtain the new offset value to apply to the delay element. Care was taken to ensure the offset values would sum to zero over an entire cycle, to prevent the nominal value of the delay from drifting. Various checks were included (not shown here for reasons of brevity) to ensure negative delay values could not be used. Each generator's start time and update rate were specified from the environment configuration, allowing for each group's updates to happen essentially asynchronously with respect to each other. This might not quite model reality, but it was deemded important to be able to stress all of the DLL's algorithms in order to uncover any corner-cases.

C. Connection and Control

The poster presentation (which will be in the DVCon archive) will have full details of the connection and control code which, due to formatting constraints, will not fit here. However, there are two use cases which can be shown.

1) Setting board delays: . Board delays are not varied in simulation as it has been determined from measurements of actual boards that they are essentially invarient, for a given board layout. However, each bytelane has a unique set of trace delays and so these are stored in the environment configuration object and then initialized through the delay control API directly at the start of

simulation. The values are set via constaints of the configuration object, which is then read and applied through the interface wrapper classes (which hold references to the concrete delay control class's APIs). The following code shows snippets of this:

Listing 6. Time-invarient delay configuration

```
//configuration object
constraint bytelane_delay_config {
  (this.board_timing_case == typical_case )
-> {
      bytelane_configs[0].dq[0] == 626;
      bytelane_configs[0].dq[1] == 626;
      bytelane_configs[0].dq[2] == 626;
      bytelane_configs[0].dq[3] == 626;
      bytelane_configs[0].dq[4] == 626;
      bytelane_configs[0].dq[5] == 626;
      bytelane_configs[0].dq[6] == 626;
      bytelane_configs[0].dq[7] == 626;
      bytelane_configs[0].dqs   == 626;
      bytelane_configs[0].dqs_b == 626;
... }
}
...
//delay control driver
function void configure_byteif_delays(int bytelane,
   time dq[8], time dqs, time dqs_b);
  byte_if_wrap.dqs_delay_api[bytelane].set_delay(dqs);
  byte_if_wrap.dqs_b_delay_api[bytelane].set_delay(dqs_b);
  for (int i=0; i<8; i++)
    byte_if_wrap.dq_delay_api[bytelane][i].set_delay(dq[i]);
endfunction
...
//delay control agent
virtual function void connect();
  for (int bytelane=0; bytelane<N; bytelane++)
   this.delay_timing_drv.configure_byteif_delays
    (bytelane,
     this.parent_cfg.delay_timing_config.bytelane_configs[bytelane].dq,
     this.parent_cfg.delay_timing_config.bytelane_configs[bytelane].dqs,
     this.parent_cfg.delay_timing_config.bytelane_configs[bytelane].dqs_b);
...
```

2) Updating dynamic delays: These updates are made via writes to each delay group's analysis port. More detail of the code will be presented in the poster.

VI. Conclusion

This paper describes how sub-cycle timing requirements for the verification of a complex DIMM system were controlled in order to model real-world variations in order to meet the designs verification objectives. The OVM implementation of the analysis pattern was used to build a flexible control mechanism for the timing configuration and control and allowed various patterns of temporal change to be applied. In concert with [3], a powerful verification environment was built to allow sub-cycle verification to be performed at the RTL level. The configuration and control mechanism described has applicability in many different verification environments where the flow of control and configuration information is separate from the normal flow of data transactions.

References

[1] Mentor Graphics and Cadence Design Systems, *Open verification methodology*, http://en.wikipedia.org/wiki/Open_Verification_Methodology, January 2008.

[2] _____, *Open verification methodology analysis ports*, http://verificationacademy.com/verification-methodology-reference/ovmworld/docs_2.1.2/html/files2/tlm_ifs_and_ports-txt.html# Analysis, 2012.

[3] Huygens, *Software analysis pattern*, http://en.wikipedia.org/wiki/Software_analysis_pattern.

[4] JEDEC, *SSTE32882 registering clock driver item 104 with parity and quad chip selects for ddr3 rdimm applications specification*, http://www.jedec.org, 2010.

[5] _____, *JESD79-3 ddr3 sdram specification*, http://www.jedec.org, 2012.

[6] Sang-hoon Kim, Se-jun; Hong and Jae-bum Ko, *Analog delay locked loop having duty cycle correction circuit*, http://www.freepatentsonline.com/7078949.html, 2006.

[7] Anders Nordstrom, *Sub-cycle functional timing verification using systemverilog assertions*, SNUG San Jose, March 2013.