Weathering the Verification Storm:

Methodology Enhancements used on a Next Generation Weather Satellite C&DH Program

Michael Donnelly
Lockheed Martin
michael.j.donnelly@lmco.com

Michael Horn
Mentor Graphics
mike_horn@mentor.com

Doug Krening
Verification Consultant
doug.krening@me.com

Abstract — **In 2010–2011, Lockheed Martin Space Systems designed and verified eight FPGAs for the Command and Data Handling (C&DH) subsystem of the NOAA/NASA Geostationary Operational Environmental Satellite R-Series (GOES-R), scheduled to launch in 2015. Hardware validation and integration of these FPGAs went smoothly. Perhaps the best measure of the success: the FPGAs performed with almost no functional failures during integration and test.**

To help with this effort, Lockheed Martin created and introduced three new parts to its verification methodology. While originally developed in OVM, these techniques have subsequently been ported to UVM. UVM versions of the enhancements will be shown and discussed here. The three enhancements are:

1. **Dramatically reducing agent development effort using a highly parameterized base agent**
2. **Providing data rate controls in a driver that preserves interesting transaction bursts and gaps while maintaining a desired throughput using an interface throttling class**
3. **Encapsulating, modifying and covering variations to an interface's pin-level timing using an interface timing class**

Keywords—UVM/OVM, FPGA, space, NASA

I. Parameterized Base Agent

When working with an agent-based UVM verification environment (an agent is an encapsulation of code needed to interact with a bus protocol), one thing becomes obvious: the architecture of most agents adheres to a small number of topologies. Though there is some variation depending on the application, an agent generally contains a monitor, sequencer, driver, configuration information and other various support classes. In each agent these classes are interconnected identically, pulling configuration and virtual interface information from the configuration database and sharing much common base code. Taking advantage of this commonality, by using vendor-specific tools to auto-generate this code, helps avoid duplication of effort.

However, these tools have two main drawbacks. First, their cost can be prohibitive. Second, the tools are relatively inflexible regarding agent architecture. That is, if an organization wishes to utilize an agent that consists of more than a sequencer, driver and monitor, it may be out of luck.

SystemVerilog's capabilities can help with these issues. Parameters combined with classes in SystemVerilog allow for the creation of a set of base classes, which together form a highly parameterized base agent. As a concrete example, consider the agent shown in Figure 1. This agent has several enhancements over the standard sequencer, driver and monitor architecture:

1. A translator port to support incoming high-level transactions in a layered protocol
2. An internal analysis block to support checking of low-level protocol-related activity within the agent.
3. Two separate analysis ports allowing the agent to separate stimulus and response transactions
4. A shared object to support variable sharing between agent members

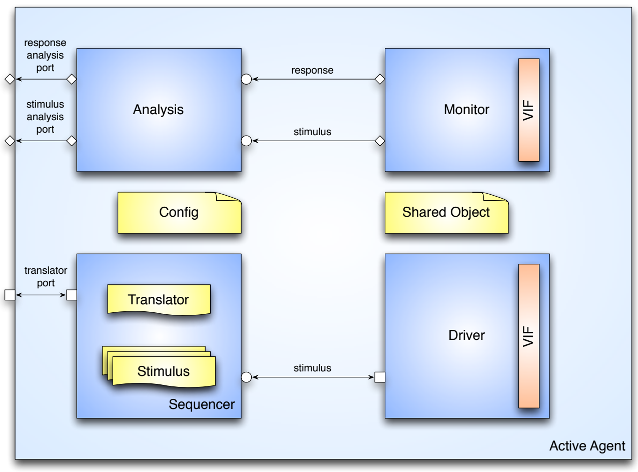


Figure 1:An ACTIVE agent architecture

The additional functionality included in this agent most likely eliminates the use of a code auto-generation tool, most of which use standard templates. Creating a reusable base agent capable of handling much of the low-level housekeeping requires diving into the world of SystemVerilog parameterized classes.

The problem at hand is to create a base class that takes advantage of the common construction of any concrete child class yet allows the types of the various class member objects to vary among its children. For instance, an RS422 agent and an AXI bus agent share the same basic architecture, yet the RS422 driver class must have radically different functionality than the AXI driver class. This problem is illustrated in Figure 2. In this figure, a base agent class attempts to declare and create a driver object. But since the driver class type must be specified in the base class, the base agent cannot instantiate a driver in a generic fashion.

```
class agent_base extends uvm_agent;
  // members
  …
  <drv_class> drv

  function void build_phase(uvm_phase phase);
    …
    drv = <drv_class>::type_id::
                            create("drv", this);
  endfunction : build_phase
  …
endclass : agent_base
```

Figure 2: Parameterized agent problem

The solution to this problem is to use the parameterized class capabilities of SystemVerilog as illustrated in Figure 3. In this figure, the type of the driver is passed into the agent as a parameter.

```
class agent_base #(type DRV = agent_drv_base)
                               extends uvm_agent;
   // members
  …
  DRV drv

  function void build_phase(uvm_phase phase);
    …
    drv = DRV::type_id::create("drv", this);
  endfunction : build_phase
  …
endclass : agent_base
```

Figure 3: Parameterized agent solution

This technique—demonstrated in Figure 4, which outlines a simplified build() method—allows for straightforward implementation of the agent base class. The agent's configuration object is first queried to determine the "flavor" of the agent; whether it is ACTIVE, PASSIVE or one of the layered agent flavors. Depending on the agent flavor, the correct components are then instantiated. In this example the types DRV, MON, ANL and SQR are all class parameters specified when creating a concrete agent. In similar fashion, all of the agent's interconnections can be handled in the base agent connect() method. In fact, all of the build and connect operations can be handled in the base agent, allowing the concrete agent designer to focus on functionality rather than housekeeping.

```
function void agent_base::
                 build_phase(uvm_phase phase);
  …
  // instantiate members based on agent type
  // defined in configuration object
  case (cfg.flavor)
    ACTIVE: begin
      resp_ap = new("resp_ap", this);
      stim_ap = new("stim_ap", this);
      mon = MON::type_id::create("mon", this);
      anl = ANL::type_id::create("anl", this);
      drv = DRV::type_id::create("drv", this);
      sqr = SQR::type_id::create("sqr", this);
    end
    PASSIVE: begin
      resp_ap = new("resp_ap", this);
      mon = MON::type_id::create("mon", this);
      anl = ANL::type_id::create("anl", this);
    end
    ACTIVE_LAYER: begin
      …
    end
    PASSIVE_LAYER: begin
      …
    end
    …
  endcase

endfunction : build_phase
```

Figure 4: Parameterized agent build() method

Prior to instantiating the agent's member components, the configuration object must be pulled from the configuration space. Retrieving objects from the configuration database requires using the path argument, which is part of the uvm_config_db API.

When using the uvm_config_db API, two pieces of information are used to get a piece of configuration information. The first is configuration name. The second piece is the UVM component path. Parameterized agents all need to pull a configuration object from the configuration database. This is handled in the base_agent code. To do this, the base agent always uses the configuration name of "config". This allows it to do a simple call to the uvm_config_db::get() as shown in Figure 5. To control which configuration object the agent receives, the test (or whichever component places the configuration object into the configuration database) must use the correct UVM component path. For example, if a testbench has an instance of a parameterized agent called "my_agent1" instantiated in an environment called "my_env", then the uvm_test would route the configuration object to "my_agent1" by performing a uvm_config_db::set() call with the path argument specified as "my_env.my_agent1*" as shown in Figure 6.

```
function void agent_base::
                    build_phase(uvm_phase phase);
  super.build_phase(phase);

  // get config object
  if(!uvm_config_db #(CFG)::get(this, "",
                                "config", cfg))
    `uvm_fatal(get_full_name(),
               "no config object available")

  // assign local VIF handle from
  // config object
  vif = cfg.vif;
  …
endfunction : build_phase
```

Figure 5: Getting the configuration object

```
function void test_base::
                    build_phase(uvm_phase phase);
  …
  uvm_config_db#(my_agent_cfg)::set(this,
                          "my_env.my_agent1*",
                          "config",
                          my_agent1_cfg);
  …
endfunction : build_phase
```

Figure 6: Test setting the configuration

We have shown that all the housekeeping functions normally performed in a concrete top-level agent can instead be performed in a parameterized agent base class. The functions consist generally of configuration database access, component instantiation and component interconnection. Since this housekeeping is generally all that a top-level agent does, it is usually possible to declare a concrete agent as a specialization of the parameterized base agent class. This is shown in Figure 7.

```
typedef agent_base #(
  .CFG (myagent_cfg),
  .VIF (virtual myagent_if),
  .TXN (myagent_txn),
  .ANL (myagent_analysis),
  .MON (myagent_monitor),
  .DRV (myagent_driver)
) myagent;
```

Figure 7: Declaring a concrete agent via typedef

Yet sometimes it is necessary to add either functionality or additional member components to the concrete agent. In these cases, it is necessary to extend the base agent in order to define the concrete agent. Figure 8 shows a simple example of this situation, where the agent needs to incorporate some reporting at the end of simulation. We move now from the base agent itself to the agent component classes.

```
class myagent extends agent_base #(
  .CFG (myagent_cfg),
  .VIF (virtual myagent_if),
  .TXN (myagent_txn),
  .ANL (myagent_analysis),
  .MON (myagent_monitor),
  .DRV (myagent_driver)
);
  `uvm_component_utils(myagent)

  // new
  function new(string name,
               uvm_component parent = null);
    super.new(name, parent);
  endfunction : new

  // report_phase
  function void report_phase(uvm_phase phase);
    …
  endfunction : report_phase
endclass : myagent
```

Figure 8: Defining a concrete agent via extension

It is also possible to perform a great deal of housekeeping functions in the base agent component classes, especially the driver. For example, we have already shown that using the path argument allows for great flexibility in configuring the parameterized base agent. This concept can be directly extended for use in the agent component classes such as the driver and monitor. As shown in Figure 9, a simple get will succeed because of the wildcard at the end of the path name in a set call.

```
function void agent_drv_base::
                    build_phase(uvm_phase phase);
  super.build_phase(phase);

  // get the cfg
  if (!uvm_config_db #(CFG)::get(this, "",
                                "config", cfg))
    `uvm_fatal(get_full_name(),
               "no config object available")
  end
  …
endfunction : build_phase
```

Figure 9: Driver's configuration database use

A very significant function that can be pushed into the driver base class is the detailed driver side of the sequence / sequencer / driver handshake. This is a constant source of trouble since the sequence and driver must agree on the protocol. If there is a mismatch between sequence and driver protocol, mayhem nearly always results. By handling this handshake in the base class, an organization can enforce code standardization and eliminate a common source of error. Figure 10 shows an implementation of both non-blocking and blocking handshakes in the driver base class.

The details of the blocking and non-blocking protocol are beyond the scope of this paper. However, note the use of the calls to the idle_cycle() and active_cycle() methods. These are empty methods to be overridden in the concrete driver class and provide the functionality of wiggling the interface pins to either consume idle time or actively drive a transaction on the bus. It is possible to make these "pure virtual" methods and to make the driver an abstract class, thereby forcing the concrete driver class to implement both methods. The driver's run phase typically calls either the blocking or non-blocking version of the process_item() task from within a forever loop.

The sequence side of the sequence / sequencer / driver handshake is much simpler but should still similarly be implemented in a sequence base class to avoid any protocol mismatch. This is illustrated in Figure 11.

```
// process_item_nb (non blocking)
task agent_drv_base::process_item_nb();
  TXN req_txn, rsp_txn;

  do begin
    seq_item_port.try_next_item(req_txn);
    if (req_txn == null) begin
      idle_cycle();
    end
  end while (req_txn == null);

  rsp_txn = TXN::type_id::create("rsp_txn");
  rsp_txn.set_id_info(req_txn);
  active_cycle(req_txn, rsp_txn);
  seq_item_port.item_done();
  seq_item_port.put(rsp_txn);

endtask : process_item_nb

// process_item (blocking)
task agent_drv_base::process_item();
  TXN req_txn, rsp_txn;

  seq_item_port.get_next_item(req_txn);
  rsp_txn = TXN::type_id::create("rsp_txn");
  rsp_txn.set_id_info(req_txn);
  active_cycle(req_txn, rsp_txn);
  seq_item_port.item_done();
  seq_item_port.put(rsp_txn);

endtask : process_item

// idle_cycle: drive idle on bus
task agent_drv_base::idle_cycle();
endtask : idle_cycle

// active_cycle: drive transaction on bus
task agent_drv_base::active_cycle(TXN req_txn,
                                  TXN rsp_txn);
endtask : active_cycle
```

Figure 10: Base driver side of SEQ/SQR/DRV handshake

```
task seq_base::put_item(REQ req_txn,
                        ref RSP rsp_txn);
  start_item(req_txn);
  finish_item(req_txn);
  get_response(rsp_txn);
endtask : put_item
```

Figure 11: Base sequence side SEQ/SQR/DRV handshake

We have shown that significant driver functionality can be implemented in a parameterized driver base class. This leaves the concrete driver designer free to concentrate on functionality alone. In general this consists of implementing the idle_cycle() and active_cycle() methods.

While there is a considerable amount of mundane code to be implemented in the agent and driver base classes, we have not identified much that can be implemented in monitor or analysis base classes. Primarily we pull objects out of the configuration database in these base classes, but leave all other functionality to the concrete classes.

There is one minor but significant problem that may trip up an organization's efforts to implement a parameterized agent methodology. The issue arises if an organization chooses to use a top-level analysis group that aggregates transactions coming from various agents' analysis ports. In this methodology, the analysis group's analysis exports must be parameterized with a common base transaction type in order to accept transactions of various types. In this case it is necessary to parameterize all the agent's analysis ports with the same common base transaction. If instead the agents use their own parameterized transaction type for the analysis ports, these ports will not connect successfully with the base transaction analysis exports.

As we have shown so far, there is a tremendous advantage to deploying a parameterized agent methodology. Much mundane, repetitive housekeeping code can be implemented once in the base classes, leaving the designer free to concentrate on functionality when designing concrete agent classes. There is, however, a significant downside. Parameterized classes are devilishly difficult to debug. Many error messages normally caught during compilation are not caught until elaboration or simulation time. When these errors appear, the error messages are typically obscure.

When creating parameterized base classes the best approach is to first develop and debug the classes without parameterization. Only when you are very confident that you have everything right should you carefully add the parameterization.

Nonetheless, once developed, a parameterized agent methodology offers significant benefits by pushing much standard housekeeping code down into the agent base classes and enforcing code standardization by ensuring all application-specific agents inherit the same base code and adhere to common design patterns.

II. DRIVER-LEVEL DATA RATE CONTROLS

In many real systems accesses occur at random intervals. To mimic this behavior, it is desirable to vary the gap between transactions in order to verify that the DUT can handle everything from back-to-back transactions to lengthy delays between transactions. In addition, randomizing the gap between transactions ensures that these transactions arrive at varying times relative to other events occurring inside the DUT. At the same time, it is often necessary to control the aggregate throughput on an interface. Throughput control allows the testbench to selectively over- or under-subscribe an interface as needed to test conditions such as FIFO underflow or overflow.

One method for simultaneously providing throughput control while maintaining interesting bursts and gaps between transactions is for the driver to utilize a throttle object to calculate inter-transaction gaps. The throttle object is used to record the number of cycles spent idling versus actively driving a transaction on the bus. Then, when the driver receives a transaction from its sequencer, the throttle object is queried in order to generate a random number of idle cycles, which should precede driving the transaction on the bus. The key innovation here is that this "number of idles" determination is made using a Poisson distribution and that the "mean" of the Poisson function is dependent on how far the measured throughput is from the desired throughput. Figure 12 through Figure 16 illustrate how the driver should utilize a throttle object.

Figure 12 shows the declaration of the throttle object in the driver's class declaration, while in Figure 13 the throttle is created from the factory and initialized with the desired throughput as specified in the agent's configuration object (cfg). It is convenient to specify throughput as an integer between 0 and 100, representing the percent of maximum interface throughput that the driver should target.

```
class my_drv extends uvm_driver #(my_txn);
  `uvm_component_utils(my_drv)

  // methods
  extern function new(string name,
                      uvm_component parent);
  extern function void build_phase(
                             uvm_phase phase);
  extern virtual task idle_cycle();
  extern virtual task active_cycle(
                              my_txn req_txn,
                              my_txn rsp_txn);
  extern virtual task process_item_nb();
  …

  // members
  throttle idle_gen;
  …
endclass : my_drv
```

Figure 12: Throttle object declaration

```
function void my_drv::build_phase(
                             uvm_phase phase);
  idle_gen = throttle::type_id::
                           create("idle_gen");
  idle_gen.set(cfg.throughput);
  …
endfunction : build_phase
```

Figure 13: Building the throttle object

The operational concept of the throttle requires that it be called throughout a simulation in order to record the number of active cycles and idle cycles consumed on the bus. This requires the sequence-sequencer-driver handshake to be non-blocking so that the driver can always track the passage of time on the bus. Consequently the driver cannot use a get() or a get_next_item() call as that would prevent the driver from recording the passage of time. Instead the driver makes use of the try_next_item() call, which does not block. An example of this handshake is shown in Figure 14. When a sequence item is not available, the driver's call to try_next_item() returns "null" and the driver calls idle_cycle() to consume one cycle of time on the bus. In turn, the throttle object's idle() method is called in Figure 15, recording the fact that one idle cycle has been consumed.

```
task my_drv::process_item_nb();

  my_txn req_txn, rsp_txn;

  do begin
    seq_item_port.try_next_item(req_txn);
    if (req_txn == null) begin
      idle_cycle();
    end
  end while (req_txn == null);

  …
  // wiggle pins
  active_cycle(req_txn, rsp_txn);
  seq_item_port.item_done();
  …
  seq_item_port.put(rsp_txn);

endtask : process_item_nb
```

Figure 14: Non-blocking Sequencer-Driver Handshake

```
task my_drv::idle_cycle();
  // record the idle cycle
  idle_gen.idle();

  // consume one cycle driving idle
  …
endtask : idle_cycle
```

Figure 15: Use of throttle object in idle cycles

If, on the other hand, the call to try_next_item() returns a sequence item the driver calls the active_cycle() method to handle the incoming transaction. As shown in Figure 16, the active_cycle() method first calls the throttle object's active() method. This method performs two functions. First, it records the number of active cycles the driver will consume in sending the transaction on the bus. The active() method also calculates the current throughput on the bus, compares this with the desired throughput, and calculates the number of idle cycles the driver should apply before sending the transaction. Following the call to the throttle object's active() method, the driver then sends the required number of idles followed by the transaction itself.

```
task my_drv::active_cycle(my_txn req_txn,
                          my_txn rsp_txn);
  int idles;

  // Calculate bandwidth throttle and
  // record active cycles
  // The transaction can override this by
  // specifying a gap
  idles = idle_gen.active(req_txn.get_size());
  if (req_txn.gap >= 0)
    idles = req_txn.gap;

  // insert idle cycles
  repeat (idles) idle_cycle();

  // consume cycles driving the txn
  …
endtask :active_cycle
```

Figure 16: Use of throttle object in active cycles

It is worth noting that a sequence can override the operation of the throttle class if it so desires. We use a base transaction class containing a member called 'gap'. This integer member directly represents the inter-transaction gap the sequence would like to apply before a transaction. This gap defaults to '-1' to indicate that the throttle class should determine the gap. However, if the gap is set to a number greater than or equal to zero, this number will be used directly rather than the throttle calculation.

Now we turn to the details of the throttle class itself. The key to understanding its operation is the Poisson distribution, used by the throttle object to calculate the number of idles to apply before a transaction. However, the mean of the distribution used is adjusted based on how far the actual throughput is from the target throughput. For reference the Poisson distribution with various values for the mean is illustrated in Figure 17.

If the actual throughput is less than the target the driver has been applying too many idles. In this case the throttle class uses a mean of 1 and in all likelihood the resultant number of idles will be less than 5. In fact the number of idles selected in this case is most likely to be either 0 or 1.

In contrast, if the actual throughput is greater than the target, the driver needs to apply more idles to decrease the throughput. In this case, the mean is chosen to equal the difference between the actual number of active cycles so far and the target number of active cycles. So, for example, if the driver has generated 20 more active cycles than desired, the mean selected is 20. On average, the throttle object will return 20 and the actual throughput at the interface will decrease accordingly.

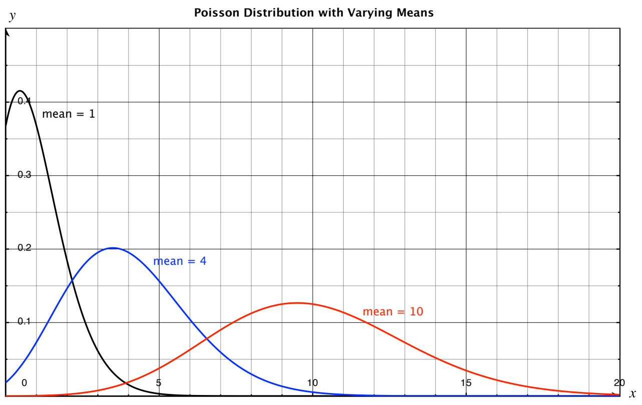


Figure 17: Example of Poisson distributions

Figure 18 shows the declaration of the throttle class. The main three class members are 'throughput', 'idles' and 'actives'. These are the target throughput and the running count of idle cycles and active cycles applied during a simulation. The class also contains a random seed for the Poisson distribution. This is randomized in the new() method and is updated with every call to the distribution function.

```
class throttle extends uvm_object;
  `uvm_object_utils(throttle)

  // methods
  extern function new(string name = "");
  extern virtual function void set(
                            int throughput);
  extern virtual function void reset(
                       int throughput = -1);
  extern virtual function void idle(
                             int count = 1);
  extern virtual function int  active(
                                 int count);
  extern virtual function int  report();

  // members
  rand int seed;

  int idles = 0;
  int actives = 0;
  int throughput = 50;

endclass : throttle
```

Figure 18: Throttle class declaration

Figure 19 and Figure 20 demonstrate some of the necessary housekeeping methods. The set() method is called when the throttle object is first created and sets the target throughput that the object will use in its calculations going forward. Recall that the target throughput specified here is a percentage of the interface's maximum possible throughput. The reset() method is not always needed. But if the bus spends significant time idling due to lack of sequencer input, this method may be used to reset the running 'idle' and 'active' counts when stimulus begins. This condition often occurs, for instance, when the bus is idling while the DUT is configured via a different interface. The reset() method may also be used to change the target throughput during a simulation. It is often convenient to add an API method to the agent which calls the reset() method.

```
function void throttle::set(int throughput);
  // initialize the target throughput
  this.throughput  = throughput;
endfunction : set
```

Figure 19: Throttle set( ) method

```
function void throttle::reset(
                         int throughput = -1);
  // conditionally set the target throughput
  if (throughput >= 0)
    this.throughput = throughput;

  // reset idle and active counts to '0'
  idles   = 0;
  actives = 0;
endfunction : reset
```

Figure 20: Throttle reset( ) method

Finally, Figure 21 and Figure 22 show the heart of the throttle class, the idle() and active() methods. As described earlier the idle() method simply records idle cycles while the active() method records the active cycles but also calculates the inter-transaction idle count based on the Poisson distribution. Note that the seed argument to the $dist_poisson() method is of type 'inout' and is updated with each call to $dist_poisson().

```
function void throttle::idle(int count = 1);
  // record idle cycle(s)
  idles += count;
endfunction : idle
```

Figure 21: Throttle idle( ) method

```
function int throttle::active(int count);
  int target;
  int mean;
  int cycles;

  // record active cycle(s)
  actives += count;

  // calculate the target number of
  // active cycles
  target = throughput * (actives+idles) / 100;

  // calculate the mean of the
  // Poisson distribution
  mean = (actives > target) ?
                      (actives - target) : 1;

  // randomize the number of idle
  // cycles needed
  return ($dist_poisson(seed, mean));
endfunction : active
```

Figure 22: Throttle active( ) method

Using this methodology, a realistic flow of transactions may easily be created that features both throughput control and burstiness. But the traffic characteristics using this technique are shared by all sequences using a particular driver. What about modeling different traffic characteristics for different sequences operating on the same driver?

Although we have not implemented it, the throttle class concept could be directly applied at the sequence abstraction level. Multiple sequences, each with its own throttle object, could easily model real world traffic patterns. For instance, one sequence could model VoIP traffic with a constant sample rate while other sequences could represent bursts of low-priority traffic. All sequences would then use the "gap" member of the base sequence item class to override the throttle class in the driver. Using this technique a testbench could readily measure the effects of bursty, low-priority traffic on the latency jitter of steady, high-priority flows.

III. Timing Controls

Frequently varying the timing parameters of an interface at the pin-level is critical to ensure the DUT meets functional requirements. In many cases, this includes not only legal interface timing but also off-nominal cases to ensure the DUT rejects and/or recovers from illegal/invalid interface timing or sequencing.

All interfaces, no matter the level of complexity, can be described by a set of timing parameters and the dependencies between these parameters. Functional verification of the interface timing is considered complete when the DUT responds correctly to all required values (or ranges of values) and important timing sequences as well as gracefully handles situations where off-nominal and/or illegal cycles occur.

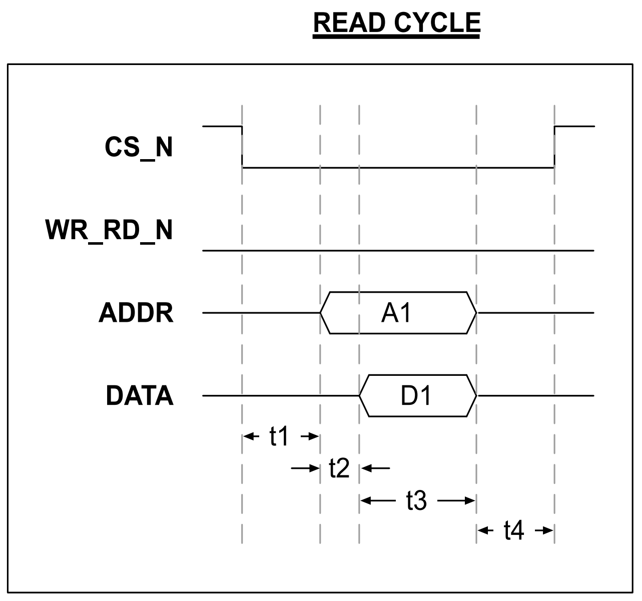
An interface timing class encapsulates the timing parameters, constraints and coverage model associated with an interface into one self-contained object. The advantages of this construct are numerous:

1. Interface timing can be modified (randomized) at any desired interval
2. Constraint-based timing definition generates legal or illegal cycle timing, upon request, for interdependent specs
3. Weighted distributions associated with these constraints ensure interesting values are generated frequently
4. Resulting agent driver code is highly succinct, readable and maintainable
5. Applicable to a broad range of interfaces, both asynchronous and synchronous

For the purpose of illustration, this paper will discuss a parallel bus with the following characteristics:

1. Active low chip select control signal
2. Read / write control signal
3. 8-bit address bus
4. 8-bit bidirectional data bus

Figure 23 shows the timing diagram for both read and write cycles on this bus. The remainder of this paper, including code snippets, will provide examples using the write functionality of this bus.



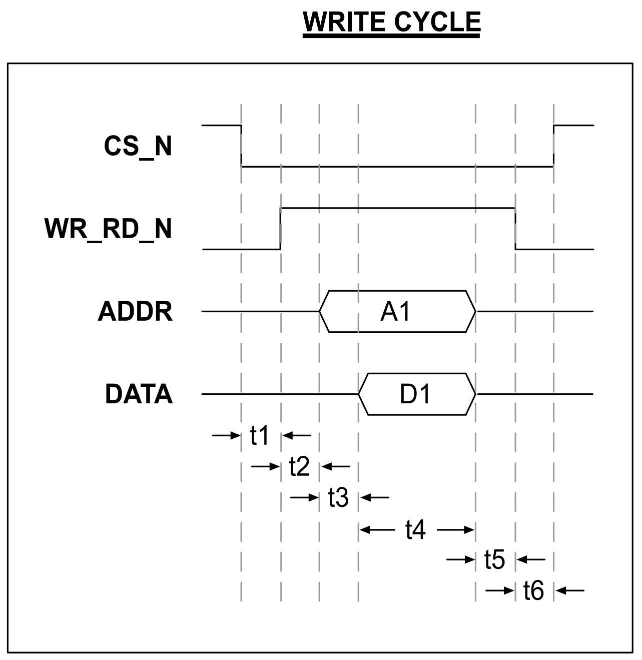


Figure 23: Bus cycle timing

Generically, an interface timing class object is comprised of four components:

1. Timing specs
2. Timing constraints
3. Timing distribution
4. Coverage model

This first component of an interface timing class is the timing parameters themselves. These typically consist of two groups of members: constants defining the bounds of each individual spec and random members utilized by the driver to perform the cycle. Figure 24 shows the declaration of the timing parameter specs and members, assigning arbitrary values to the limits of each parameter.

Two coding styles exist: the use of SystemVerilog parameters (or localparams) and the use of constant ints. Using parameters is more efficient from the perspective of memory footprint, but introduces the possibility of conflicts if a generic naming convention is reused across multiple timing classes. The provided example makes use of constant ints to avoid these potential namespace issues.

```
//=============================================
// write cycle timing specs, in ns
//=============================================
const int W1_MIN    = 100;
const int W1_MAX    = 200;

const int W2_MIN    = 150;
const int W2_MAX    = W2_MIN*2;

const int W3_MIN    = 100;
const int W3_MAX    = W3_MIN*2;

const int W4_FIXED  = 300;

const int W5_MAX    = 200;
const int W5_MIN    = W5_MAX/2;

const int W6_MAX    = 200;
const int W6_MIN    = W6_MAX/2;

//=============================================
// timing specs, randomized each cycle
//=============================================
rand int w1, w2, w3, w4, w5, w6;
```

Figure 24: Timing parameter definition

The second component of an interface timing class is the constraints defining the bounds and interdependencies between parameters. Figure 25 shows the declaration of the timing constraints for the write cycle parameters. The majority of the constraints bound the individual timing specifications within their legal ranges. The last two constraints define dependencies between two or more parameters. In this example, there are two cases where overlapping ranges require additional definition. First, the chip select signal must assert before the write/read control signal asserts, described by the w1_w2_phase constraint. At the end of the cycle, the write/read control must deassert before chip select deasserts, described by w5_w6_phase.

Note that the timing variables, w1 through w6, are type int in order to mesh with the functional coverage model. Neither real nor time types can be directly sampled in coverage and must be converted. To avoid this step, the timing variables themselves are unitless members of type int. Both the coverage model and agent driver are written to scale the integer values as needed to provide the required level of resolution.

```
//=============================================
// write cycle timing constraints
//=============================================
constraint w1_range
           { w1 inside {[W1_MIN : W1_MAX]}; };
constraint w2_range
           { w2 inside {[W2_MIN : W2_MAX]}; };
constraint w3_range
           { w3 inside {[W3_MIN : W3_MAX]}; };
constraint w4_fixed
           { w4 ==  W4_FIXED; };
constraint w5_range
           { w5 inside {[W5_MIN : W5_MAX]}; };
constraint w6_fixed
           { w6 inside {[     0 : W6_MAX]}; };

constraint w1_w2_phase  { w1 < w2; };
constraint w5_w6_phase  { w5 < w6; };
```

Figure 25: Timing constraint definition

The third component of an interface timing class is a set of weighted distributions to shape randomization outputs. Figure 26 shows the weighted distributions associated with the timing constraints. Although these distributions are optional from a purely functional viewpoint, they ensure that interesting / bounding values are generated at a relatively high rate. In practice, explicitly defined min/max bins correspond to those timing specs with explicitly defined bounds. Derived bounds, such as W2_MAX = 2*W2_MIN, are not assigned specific bins. Note that this example uses hard-coded weights for all distributions. More complex interfaces may require the flexibility of variable-controlled weights; in practice, however, hard-coded weights produce an acceptably diverse variety of timing.

```
//=============================================
// write cycle timing distribution
//=============================================
constraint sc_w1_dist  { w1 dist {
                      W1_MIN := 10,
                      [W1_MIN : W1_MAX] :/ 80,
                      W1_MAX := 10 }; }
constraint sc_w2_dist  { w2 dist {
                      W2_MIN := 10,
                      [W2_MIN : W2_MAX] :/ 80,
                      W2_MAX := 10 }; }
constraint sc_w3_dist  { w3 dist {
                      W3_MIN := 10,
                      [W3_MIN : W3_MAX] :/ 80,
                      W3_MAX := 10 }; }
// w4 is fixed, no dist necessary
constraint sc_w5_dist  { w5 dist {
                      W5_MIN := 10,
                      [W5_MIN : W5_MAX] :/ 80,
                      W5_MAX := 10 }; }
constraint sc_w6_dist  { w6 dist {
                      [0 : W6_MAX] :/ 75,
                      W6_MAX := 25 };}
```

Figure 26: Timing Constraint Distributions

The final component of an interface timing class is the coverage model. Figure 27 shows the coverage model for the write cycle of the 8-bit example bus. The coverage model looks nearly identical to the weighted distributions described previously. The covergroup explicitly defines min/max for those timing parameters with explicit bounds. Range bins are then defined for all non-fixed parameters to ensure a variety of values are tested for each parameter. Cross coverage can be added, as required, to cover timing occurrences between interdependent specs/signals.

```
//=============================================
// write cycle timing coverage
//=============================================
covergroup wr_cvg();
  w1_cvp: coverpoint w1 {
   bins min              = {W1_MIN};
   bins range[RNG_BINS]  = {[W1_MIN : W1_MAX]};
   bins max              = {W1_MAX}; }
  w2_cvp: coverpoint w2 {
   bins min              = {W2_MIN};
   bins range[RNG_BINS]  = {[W2_MIN : W2_MAX]};
   bins max              = {W2_MAX}; }
  w3_cvp: coverpoint w3 {
   bins min              = {W3_MIN};
   bins range[RNG_BINS]  = {[W3_MIN : W3_MAX]};
   bins max              = {W3_MAX}; }
  // no cvp for w4, it is fixed
  w5_cvp: coverpoint w5 {
   bins min              = {W5_MIN};
   bins range[RNG_BINS]  = {[W5_MIN : W5_MAX]};
   bins max              = {W5_MAX}; }
  w6_cvp: coverpoint w6 {
   bins min              = {W6_MIN};
   bins range[RNG_BINS]  = {[W6_MIN : W6_MAX]};
   bins max              = {W6_MAX}; }
endgroup: wr_cvg
```

Figure 27: Write cycle coverage model

Now that the interface timing class is complete, let's examine the resulting agent driver code that takes advantage of it. Figure 28 shows the driver's task for executing a write cycle. Note that this code skips over the instantiation and creation of the timing class object as well as assignment of the virtual interface handle. By exploiting an interface timing class, the agent driver code is extremely easy to code and "reads" like a description of the timing diagram. This simplifies code development and maintenance. Modification of the interface timing can be performed as frequently as desired, typically once per cycle as shown. After randomizing the timing for a given cycle, the driver merely executes a sequence of waits and signal assignments to drive the cycle to the DUT. The coverage model is sampled at the same rate of timing class randomization to cover the permutations executed.

As previously discussed, this example uses unitless timing variables of type int. The agent driver code interprets the timing variables and "scales" them as needed. In this example, the scale of the timing variables is assumed to be the same as the timescale of the driver itself. If required, additional resolution can be provided by enlarging the int values in the timing class itself, then scaling the values within the agent driver using local members of type real or time.

```
//==============================================
// wr_cycle
// translate write txn to pin wiggles
//==============================================
task drv::wr_cycle();
  //Randomize timing for this cycle
  if(!t.randomize())
    `uvm_error("time spec object",
                                  "randomization error")
  //initiate cycle
  vif.cs_n = 'b0;
  #(t.w1);
  vif.wr_rd_n = 'b1;
  #(t.w2);

  //send addr and data
  vif.addr = req_txn.addr;
  #(t.w3);
  vif.data = req_txn.data;
  #(t.w4);

  //terminate cycle
  vif.wr_rd_n = 'b0;
  #(t.w5);
  vif.cs_n = 'b1;
  #(t.w6);

  //collect coverage
  t.wr_cvg.sample();
endtask :wr_cycle
```

Figure 28: Driver Utilizing Interface Timing Class

All examples up to this point illustrate an interface timing class generating legal cycle timing and sequencing. This technique can be extended to generate illegal timing in order to ensure the DUT appropriately handles off-nominal cases. Depending on the complexity of the interface, two coding styles exist for generating illegal timing.

Both legal and illegal timing can be described in a single interface timing class. Using this approach, one or more flags are created to control which timing parameter(s) or sequencing will be generated illegally. The timing constraints and distributions shown in Figure 25 and Figure 26 are modified to use a conditional constraint format that is dependent on the appropriate flag. A new covergroup for illegal timing is written and the driver samples this covergroup when using illegal timing instead.

More complex interfaces may require a second interface timing class for illegal timing and sequencing. Using this approach, the constant timing parameter specs will likely need to be moved to a common package that both the legal and illegal timing classes can access. The illegal class then mirrors the structure of its legal counterpart but updates the constraints, distributions and coverage to describe off-nominal cases. The driver instantiates both a legal and illegal timing class, then randomizes, uses and samples the appropriate one as dictated.

Using either approach, the same updates must be made and the resulting amount of new code is approximately the same. The user is trading off the convenience of having all timing information, both legal and illegal, in a single, self-contained object against code readability and total lines of code per class. Depending on the ordering, the driver may need specialized tasks to execute cycles with an illegal order of events.

The advantages of using an interface timing class are numerous:

1. Interface timing can be modified (randomized) at any desired interval
2. Constraint-based timing definition generates legal or illegal cycle timing, upon request, for interdependent specs
3. Weighted distributions associated with these constraints ensure interesting values are generated frequently
4. Resulting agent driver code is highly succinct, readable and maintainable
5. The technique is applicable to a broad range of interfaces, both asynchronous and synchronous

First, the frequency at which the interface timing is modified is easily controlled and can be set to any desired interval. Depending on the interface being modeled, timing can be modified mid-cycle, once per cycle, once per simulation or anything in between. This adds tremendous flexibility to the test environment and ensures the DUT sees a robust mixture of timing on each interface.

Second, describing the interface using a complete set of constraints ensures that legal (or illegal, if desired) timing and sequencing is generated for interdependent specs. A single randomization call in the agent driver is all that is required to modify the behavior of the interface, regardless of its complexity.

Third, utilizing weighted distributions ensures that bounding or corner case values are generated with relatively high frequency. Typically hard-coded weights provide acceptable coverage, but the technique can be extended to exploit variable-controlled weights if the complexity of the interface requires. Generating absolute min/max values often likewise ensures the DUT meets requirements for these explicit bounds.

Fourth, the agent driver code which utilizes the timing class is highly succinct, readable and maintainable, as shown previously in Figure 28. An interface timing class helps logically separate the generation of timing information from the execution of bus cycles. As a result, both code for the timing class as well as the driver is specific and compartmentalized, adhering to good OOP principles. Agent driver code for wiggling pins is very straightforward and "reads" like a timing diagram. This code is easy to write and, perhaps more importantly, easy to maintain.

Finally, interface timing classes apply to a broad range of interfaces, both synchronous and asynchronous. The example illustrated in this paper focused on a simple asynchronous interface. Though beyond the scope of this paper, it is not difficult to extend the same example to demonstrate synchronous interface behavior. An interface clock is added to the timing shown in Figure 23. Existing timing parameters can be modified or redefined as timing relative to clock edges. In addition, new parameters can be added to specify setup and hold criteria relative to the clock, if necessary.

IV. Conclusion

With parameterized agents, driver level data rate controls and timing controls in addition to utilizing the broad capabilities of OVM and UVM, the GOES-R C&DH team produced FPGAs that performed admirably in the lab. This resulted in overall reduced schedule time and happy program managers. The three techniques outlined above have a wide application space and hopefully will save other engineers and programs some time and effort.

V. Acknowledgements

Thank you to Don Hewitt for being a sounding board for many of these ideas and to Geoff Koch for editing and formatting help.

VI. References

[1] IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language, IEEE Std. 1800-2009.

[2] Universal Verification Methodology (UVM), http://www.accellera.org/activities/committees/vip/