Shutdown with Agreements in a UVM Testbench

Mark Glasser

NVIDIA Corporation

Santa Clara, CA, USA

nvidia.com

ABSTRACT

Shutting down a testbench correctly and securely is not as simple as just calling $finish after the last stimulus has been generated. It is important to ensure that all the stimuli is processed and the Design under Test (DUT) is allowed to respond. The testbench must find a safe time to shutdown, when both the testbench and the DUT are quiescent*.*

This paper discusses the notion of quiescence in a testbench and DUT and how to find quiescent points using a shared synchronization primitive. Entities such as monitors, drivers, and sequences use the shared synchronization primitive to vote for or against quiescence. When all participating entities agree that the system is in a quiescent state then, by definition it is, and the testbench can safely shutdown.

The shared synchronization primitive is an object called an agreement *which is derived from* `uvm_barrier`*. The agreement API and use models are discussed in detail.*

Table of Contents

1. Introduction

Terminating a simulation is not just a matter of calling $finish (in a Verilog simulation). That is too heavy handed and leads to incomplete transactions and questionable results. Before terminating a testbench it is important to ensure that all the stimuli has been processed and both the design an the testbench are in a *quiescent* state. You need to be sure that all of the stimuli have completed and that the associated responses have also completed. In this paper we will define a methodology in UVM for shutting down testbenches safely and securely.

First, we need to introduce some concepts that pertain to shutdown.

Quiescence: Quiescence, or the state of being quiescent, refers to something that is inactive or at rest. The meaning when applied to a testbench is similar. A DUT or a testbench is quiescent if there are currently no transactions, instructions, or interrupts in flight. That does not necessarily mean it is not doing anything at all. Clocks can be running or watchdog timers can be timing, for example.

Shared Object: As the name implies, a shared object is an object that is shared amongst multiple entities in a program. Typically the entities are separate, non-overlapping scopes. A shared object is distinct from a global object. A global object resides in the global namespace and is visible by all scopes. By contrast a shared object is not in the global namespace and is only visible in certain scopes. In UVM, the resource database is a convenient medium for sharing objects.

2. When to Shut Down

The first order of business is to determine when to shut down the simulation. Essentially, it's time to finish the simulation when the available stimuli are exhausted, or when some event occurs, AND the device has reached quiescence. Let's first consider a simple case of a stimulus generator in a UVM sequence:

```
task body();

  for(int i = 0; i < 100; i++) begin
    item = new();
    start_item(item);
    item.randomize();
    finish_item(item);
  end

endtask
```

In the code fragment above, a sequence `body()` task simply generates 100 randomized transactions. It has no knowledge of what happens to each transaction after `finish_item()` is called. As far as the sequence is concerned, each sequence item (transaction) has been sent and that's all that it cares about. The sequence doesn't know how long it will take to process each transaction, so it cannot completely control the shutdown. The best it can do is to issue a notification that its part is done.

Consider another case where we have highly randomized stimuli and want to shut down when a certain register has been covered.

```
bit [5:0] instr_reg;
```

```
covergroup cg;
  instr: coverpoint instr_reg;
endgroup
```

When `instr_reg` has contained all of its possible values at least once then we want to terminate; when the following expression evaluates to true (or 1):

```
cg1.get_inst_coverage() == 100
```

At the exact point that the coverage condition becomes true we don't know anything else about the state of the system, only that one particular register has been covered. So this entity also cannot completely control the shutdown, at least not by itself. Like then stimulus generating sequence, the responsibility of the entity evaluating this condition is to issue some sort of notification that, as far as it's concerned, we are done.

It's easy to imagine more complex scenarios where multiple streams of traffic are flowing into the DUT through different interfaces. We will want to shut down when the DUT has reached quiescence; when we are "between" transactions on all interfaces. Each entity involved in the testbench only knows what is going on within its boundaries and can only say when its part is done. None of them individually can definitively say that the entire testbench should shut down.

3. Voting

There may be one or many entities in a testbench that have a vested interest in when the testbench shuts down. For example, sequences know when stimulus is complete; scoreboards know if there are any expected responses that have not yet arrived; monitors know if transactions are still in flight; coverage collectors know if specific coverage items have not yet been covered. Each of these entities must be able to vote on whether or not it's time to shut down based on their limited view of the world. When all participating entities vote that it's OK to shut down at the same time, then we have achieved quiescence and the simulation can safely shut down.

Voting is accomplished through a shared object. Each participating entity has access to the shared object, and registers its vote every time something of interest occurs in the entity that would change its vote. In our simple sequence, we would vote to NOT shutdown prior to starting stimulus generation, and vote to shut down after the stimuli have all been generated.

We place calls to register the vote at points when we know it's OK to shut down or not. For example, in our sequence stimulus generator we would vote to not shutdown immediately before we start generating stimulus, and we would vote that it's OK to shutdown as soon as all the stimulus has been generated and successfully sent.

```
task body();

  vote_no_shutdown();

  for(int i = 0; i < 100; i++) begin
    item = new();
    start_item(item);
    item.randomize();
    finish_item(item);
  end
```

```
  vote_ok_to_shutdown();

endtask
```

An entity can change its vote back and forth as things move along, like this scoreboard:

```
task run_phase(uvm_phase phase);

  transaction_t t;

  forever begin
    vote_ok_to_shutdown();
    input_fifo.get(t);
    vote_no_shutdown();
    process_transaction(t);
  end

endtask
```

Before a transaction appears on the input fifo it's OK to shut down. Once a transaction is retrieved by `get()`, then it's not OK to shut down until the transaction has been processed. Then we can return to the quiescent state and wait for a new transaction.

4. Barriers

UVM provides an object that works well as a shared object for voting. It's called a *barrier*. Barriers were initially designed to synchronize multiple process threads. We will use them a little differently to manage voting.

First, let's look at how barriers work. The barrier API includes a blocking task, `wait_for()`, which each process can use to establish a synchronization point. The function `set_threshold()` identifies the number of process that are waiting, known as *waiters*, that are to be synchronized. Each process calls `wait_for()` to include itself in the set of waiting processes. `Wait_for()` blocks until the number of *waiters* reaches the established threshold. Then they all simultaneously unblock.

As an example, consider three processes represented by three tasks, t1(), t2(), and t3().

```
task t1();
  b.set_threshold(b.get_threshold() + 1);
  // do some stuff that consumes time
  b.wait_for();
  // do more stuff
endtask

task t2();
  b.set_threshold(b.get_threshold() + 1);
  // do some stuff that consumes time
  b.wait_for();
  // do more stuff
endtask
```

```
task t3();
  b.set_threshold(b.get_threshold() + 1);
  // do some stuff that consumes time
  b.wait_for();
  // do more stuff
endtask
```

The three tasks are spawned in three processes:

```
fork
  t1();
  t2();
  t3();
join
```

For the purposes of this example, let's assume that the part of each task informally labeled "do some stuff that consumes time" consumes a different amount of time in each process. So one of the processes will finish first, another second, and the third one will finish last. We do not know in advance how much time each one will consume, so we do not know the order in which they will complete, or more specifically, the order in which they will reach the `wait_for()` call.

As each task starts, it increments the threshold using the `set_threshold()`. The current threshold is retrieved from the barrier, incremented, and the new value is set as the threshold. This tells the barrier that another process is contributing to the synchronization. Each process increments the threshold as it begins execution. The threshold will reach its maximum value --- in our case 3 -- after all the participating processes are launched. Each process will then do whatever it's going to do and eventually end up at the `wait_for()` call. The first two processes to reach the `wait_for()` call will block. When the third one calls `wait_for()`, the task will see that the number of processes calling `wait_for()` is equal to the threshold, 3, so it will not block. Immediately the other two blocked processes will unblock. Thus the three processes are synchronized at the point where each calls `wait_for()`. Once all three process have reached `wait_for()`, they will all unblock and begin the "do more stuff" part of the task.

To make the `uvm_barrier` serve as a voting object instead of as a synchronization object, we will use it a bit differently. Instead of multiple processes calling `wait_for()`, there is only one location where it is called. It is called just before the point where the shutdown is initiated. As soon as the `wait_for()` task returns then the shutdown process begins.

Voting is accomplished by manipulating the threshold, so there will be many places where `set_threshold()` is called. To vote that it is *not* OK to shutdown, increment the threshold; to vote that it is OK to shutdown, decrement the threshold.

Our simple sequence now looks like this:

```
task body();

  // increment threshold
  barrier.set_threshold(barrier.get_threshold() + 1);

  for(int i = 0; i < 100; i++) begin
```

```
    item = new();
    start_item(item);
    item.randomize();
    finish_item(item);
  end

  // decrement threshold
  {barrier.set_threshold(barrier.get_threshold() - 1);

endtask
```

Voting is accomplished by incrementing and decrementing the threshold of a barrier. A single call to `wait_for()` in the test or the top-level environment is used to determine when it's time to shutdown.

```
class test extends uvm_component;

  uvm_barrier ok_to_shutdown;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  . . .

  task build_phase(uvm_phase phase);
    ok_to_shutdown.set_threshold(1);
  endtask

  task run_phase(uvm_phase phase);

    // run sequences
    ok_to_shutdown.wait_for();

  endtask

endclass
```

In the code example above, there is a barrier named `ok_to_shutdown` which we will use as the shared object for determining when to shut down the simulation. In the build phase, we set the threshold to 1. It's important that we set it to a known value and not simply increment it, because otherwise we don't know what the value is. It may have been set somewhere else. Since `build_phase()` is a top-down phase we know that the `build_phase()` task in the component at the top of the component hierarchy will run first. Thus we are assured that we are not conflicting with any other processes that increment the barrier's threshold. The `run_phase()` task contains the only call to `ok_to_shutdown.wait_for()` in the testbench. When all the processes have voted that it's OK to shut down, the threshold will be 1, which matches the initial threshold. At that point the simulation will terminate.

5. Safety

The mechanism we've shown so far can be used to manage shutdown in a real testbench. However there are some corner cases that must be accounted for to ensure that the mechanism will always

work correctly, no matter the situation. We are relying on a number of testbench entities to correctly manipulate a shared `uvm_barrier` to register their vote to shut down or continue. What if the code that does the voting is not quite correct? The testbench could shut down prematurely if some entity incorrectly votes to allow shutdown. Or, conversely, the simulation could hang if some entity incorrectly continues to vote to disallow shutdown. Neither case is good. The latter case is harder to debug since the simulation has to be killed indiscriminately. To avoid simulations hanging we need a fail safe timeout. The fail safe time out will cause the testbench to shut down after some suitably long time has elapsed no matter how the votes are tallied. Here's an update to our `run_phase()` task in the top-level environment:

```
task run_phase(uvm_phase phase);

    uvm_sequence_base seq;

    phase.raise_objection(this);

    fork

      begin
        #50000;
        `uvm_error("TIMEOUT", "simulation timed out");
      end

      begin
        if(uvm_resource_db#(uvm_sequence_base)::
           read_by_name(get_full_name(),
                        "main_seq",
                         seq, this))
          seq.start(axi_master.mst_sqr);
        else
          `uvm_warning("NO SEQ",
            "No sequence to drive the AXI master");
        $display("waiting for barrier...");
        ok_to_shutdown.wait_for();
      end

    join_any

    disable fork;

    phase.drop_objection(this);

  endtask
```

We fork two processes, one that executes the main test sequence and another that is simply a timeout. Either the test sequence will finish properly or the timeout will trigger. In either case the simulation is guaranteed to shut down. The amount of time before timeout occurs, 50,000ns in the example, is arbitrary. It should be long enough to ensure that the simulation completes without timeout if everything works correctly, but not so long that the simulation runs for a long time once things hang up. Note that the `ok_to_shutdown.wait_for()` call is in the process that runs the sequences. The timeout processes doesn't really care about niceties like barriers. If the timeout triggers then a shutdown occurs without any further ado. However, the process that runs the sequence does care about an orderly shutdown.

6. Sharing

The barrier used for tallying votes has to be shared amongst entities that are participating in the vote. In the general case these entities have separate scopes and do not share data. There are two means of sharing barriers in UVM. Either is acceptable. One is the resources database. The top-level environment can create a barrier object and then put it into the resources database. The resource database is globally visible, so any entity can retrieve things from it, including barriers. Another way is to use a global pool. UVM has a base class called `uvm_pool#(KEY, T)`, which is a dictionary of type T whose members can be stored and retrieved via a key whose type is KEY. A pool to store barriers it typically declared as:

```
uvm_pool#(string, uvm_barrier)
```

Specializations of `uvm_pool#()` are singletons. A handle to a singleton pool can be retrieved with the static function `get_global_pool()`. In the constructor of the top-level environment component we create a barrier and put it in the barrier pool like this:

```
function new(string name, uvm_component parent);
   uvm_barrier barrier;
   uvm_pool#(string, uvm_barrier) pool;

   super.new(name, parent);

   barrier = new("ok_to_shutdown");
   pool = uvm_pool#(string, uvm_barrier)::get_global_pool();
   pool.add(barrier.get_name(), barrier);
 endfunction
```

From this point forward, each entity wishing to participate in the shutdown vote can get a handle to barrier through the barrier pool.

```
   ok_to_shutdown = uvm_pool#(string,
   uvm_barrier)::get_global("ok_to_shutdown");
```

The only thing is that all the entities need to agree on the name of the barrier, in this case "ok_to_shutdown", so they can be sure they are all sharing the same barrier.

7. Agreements

The term *agreement* is based on terminology developed in [1]. That text discusses reaching consensus among distributed processes by reaching an agreement. An agreement in our context is a consensus among multiple threads. We are using it to initiaite shutdown, but it could also be used in any situation where multiple threads are to reach consensus.

The `get_threshold()/set_threshold()` API is a bit clumsy. It's somewhat verbose, and more importantly, it's not clear that incrementing and decrementing the threshold is the means for registering a vote. We can mitigate the clumsiness of the `uvm_barrier` API by creating a subclass of `uvm_barrier` called `agreement`. The `agreement` is derived from `uvm_barrier` and has an API that is much simpler and more intuitive. We can also add some debugging features that are otherwise missing in the `uvm_barrier` class.

Here is the API for the `agreement` class:

```
class agreement extends uvm_barrier;

  function new(string name = "agreement");
  static function void set_debug();
  static function void clr_debug();
  function void agree(uvm_object obj = null);
  function void disagree(uvm_object obj = null);
  task wait_for_agreement(uvm_object obj = null);
  function void clear(uvm_object obj);
  local function string status_msg(uvm_object obj = null);

endclass
```

The key methods in the API are `agree()`, `disagree()`, and `wait_for_agreement()`. `Agree()` decrements the barrier threshold, and `disagree()` increments it which is how votes are registered. `Wait_for_agreement()` is simply a wrapper around `uvm_barrier::wait_for()`. The static function `set_debug()` turns on a debug flag that causes a variety of messages to print when using agreements. These are helpful to see which processes are exercising the agreements to determine which one may be errant. The `clr_debug()` method turns off the debug flag and stops the messages. The debug flag is off by default. The `status_msg()` method returns a string with the name of the `uvm_object` passed in as an argument along with the current threshold setting of the agreement. If no object is passed in, then the object name is omitted from the message. The function does not print the message, it only returns a string. You can print it with the UVM message functions.

Replacing barriers with agreements, our simple sequence becomes:

```
task body();

  ok_to_shutdown.disagree();

  for(int i = 0; i < 100; i++) begin
    item = new();
    start_item(item);
    item.randomize();
    finish_item(item);
  end

  ok_to_shutdown.agree();

endtask
```

Instead of a barrier pool, we obtain the agreement from an agreement pool. The technique is similar.

```
  function new(string name, uvm_component parent);
    agreement agmt;
    uvm_pool#(string, agreement) pool;

    super.new(name, parent);
```

```
    agmt = new("ok_to_shutdown");
    pool = uvm_pool#(string, agreement)::get_global_pool();
    pool.add(agmt.get_name(), agmt);
  endfunction
```

8. Voting Multiple Times

Besides being clumsy, one of the limitations with the barrier API is that the only operations you have available is to change the threshold for waiters. It can be difficult to match every increment operation with a decrement operation. If the number of decrements does not exactly match the number of increments, then the `wait_for()` task will never unblock, causing the simulation to hang or to timeout.

The agreement API avoids this problem by allowing each entity participating in the vote to only register their vote once. Repeated votes of the same kind from the same entity do not change the results. An agreement does not count the threshold level, instead it tracks votes by entity. When all entities have agreed, then the `wait_for()` task call will unblock. The one-entity-one-vote system is much easier to implement in a testbench and certainly easier to debug.

9. The Right to Vote

Some thought needs to go into deciding which entities should participate in the vote on whether or not to shut down the simulation. You should be careful to make the voting process as straightforward as possible. It's advantageous to avoid unnecessary complexity which is the root of all bugs. Here are the kinds of entities that have knowledge that can contribute to determining shutdown.

9.1 Sequences

Sequences generate the stimuli and therefore know when there is no more to generate in the case where there is a finite termination of stimuli. Sequences that receive responses through `get_response()` (or the asynchronous response handler) know when all the required responses have been returned.

9.2 Scoreboards

Scoreboards operate in terms of matching actual results to expected results. Thus, they know what is expected. The scoreboard can vote to prevent shutdown when waiting for as-yet unmatched expectations.

9.3 Monitors and Drivers

Monitors and drivers know the precise state of each transaction, including partially completed transactions. They can prevent shutdown when transactions are incomplete. To precisely determine quiescence it is important to enable monitors and drivers to vote.

It's not always possible to instrument monitors with agreements for voting. We don't always know the name of the barrier a priori, or we may have verification IP (VIP) that was built without this particular shutdown mechanism in mind. Further, we may not even have access to the source code

for some protocols. A nice way around this is to insert callbacks at key state transition points for transactions. Or, you can lobby your Favorite VIP Vendor to do the same. We can use callbacks to manipulate the agreements.

```
class ahb_monitor_callbacks
  extends ahb_monitor_base_callbacks;

    uvm_barrier ok_to_shutdown;

    function new();
      super.new();
      ok_to_shutdown = uvm_pool#(string, uvm_barrier)
        ::get_global("ok_to_shutdown");
    endfunction

    virtual function void pre_write_trans();
      ok_to_shutdown.disagree();
    endfunction

    virtual function void post_write_trans();
      ok_to_shutdown.agree();
    endfunction

    virtual function void pre_read_trans();
      ok_to_shutdown.disagree();
    endfunction

    virtual function void post_read_trans();
      ok_to_shutdown.agree();
    endfunction
endclass
```

The constructor of the callback object retrieves the shared barrier. Each callback function either increments or decrements the barrier to indicate whether it's OK to shut down or not. Using this technique we allow shutdown only when any AHB transactions that were initiated are complete.

10. What about Objections?

UVM has another object that is intended to be used as a shared synchronization primitive. It's called `uvm_objection`. The objection object can be used in the same manner as agreements to orchestrate shutdown. So why not just use objections instead of inventing yet another class?

We take a less-is-more approach to programming. Programs of any sort should use only the resources required to do the job and no more. Objections are heavyweight objects that provide lots of unnecessary services. By unnecessary we mean unnecessary to the task of serving as a shared synchronization primitive. They also impose a heavy overhead burden.

Objections manage something called *drain time*. Drain time is an arbitrary, user defined delay between when the call is made to drop an objection and when it is finally dropped. The idea is that

this delay allows time for the DUT and testbench to process any remaining activity in the pipeline and come to a quiescent point. The problem is that the delay is, well, arbitrary. No matter what delay the user picks it is likely to be wrong. If the delay is too short then all of the activity will not drain. If the delay is too long there is a risk that some activity will start somewhere and the DUT and testbench will no longer be quiescent. There is no way to tell what exactly the delay should be.

To manage the drain time delays the objection object in UVM contains a mini process scheduler. The scheduler is started when `uvm_root::run_test()` is first called. An objection context entry is scheduled to run each time a drain time is set for an objection. The process management includes watching for re-raising the objection before the drain time elapses. A lot of overhead is devoted to process management within the objection object. The overhead includes forking processes and allocating memory. All of these are very expensive and entirely not necessary.

The objection object also has the dubious feature of hierarchical propagation. This is a feature that enables an objection in a parent component to be notified if an objection in a child component changes state. This is another very expensive and unnecessary operation.

To manage synchronization across separate processes all that is required is a shared integer that can be incremented, decremented, and tested. The `uvm_barrier` provides a shared integer. The `agreement` API provides a straightforward use model. The expensive overhead in `uvm_objection` does not contribute to the required functionality.

11. Conclusion

Using a shared object for voting on whether or not to shut down a simulation provides a fine-grained means of determining when to shut down and allows for the termination process to be orderly. An orderly shutdown guarantees that the tests executed during simulation are completed properly and their results are valid.

12. Acknowledgments

The author would like to thank Cliff Cummings of Sunburst Design for the encouragement to publish this paper. Also we would like to thank Anshu Nadkarni and Darrell Boggs of NVIDIA for their support and review of this paper.

Agreement Implementation

The following is a complete implementation of the agreement class.

```
class agreement extends uvm_barrier;

  typedef enum{AGREE, DISAGREE, WAIT} agree_t;

  local static bit debug = 0;
  local static agree_t participants[uvm_object];

  function new(string name = "agreement");
```

```
    super.new(name);
    set_threshold(1);
  endfunction

  static function void set_debug();
    debug = 1;
  endfunction

  static function void clr_debug();
    debug = 0;
  endfunction

  local function void update(uvm_object ptcpnt,
                             agree_t a);

    // If there is no change in vote,
    // then there is nothing to update
    if(participants.exists(ptcpnt) &&
       participants[ptcpnt] == a)
      return;

    participants[ptcpnt] = a;

    case(a)
      AGREE:    set_threshold(get_threshold() - 1);
      DISAGREE: set_threshold(get_threshold() + 1);
      WAIT:; // do nothing
    endcase

  endfunction

  function void agree(uvm_object obj,
                      int lineno = 0);
    update(obj, AGREE);
    if(debug)
      `uvm_info("AGREE",
                status_msg(obj, lineno),
                UVM_NONE);
 endfunction

  function void disagree(uvm_object obj,
                         int lineno = 0);
    update(obj, DISAGREE);
    if(debug)
      `uvm_info("DISAGREE",
                status_msg(obj, lineno),
                UVM_NONE);
  endfunction

  task wait_for_agreement(uvm_object obj,
                          int lineno = 0);
    wait_for();
    if(debug)
      `uvm_info("ALL IN AGREEMENT",
                status_msg(obj, lineno),
                UVM_NONE);
  endtask
```

```
  function clear(uvm_object obj,
                 int lineno = 0);
    set_threshold(1);
    if(debug)
      `uvm_info("CLEAR",
                status_msg(obj, lineno),
                UVM_NONE);
  endfunction

  local function string status_msg(uvm_object obj = null,
                                   int lineno = 0);

    string msg;

    case(1)
      (obj == null) && (lineno == 0) :
          $sformat(msg, "[%s] threshold = %0d",
                        get_name(), get_threshold());
      (obj == null) && (lineno != 0) :
          $sformat(msg, "[%s] threshold = %0d @ %0d",
                         get_name(), get_threshold(), lineno);
      (obj != null) && (lineno == 0) :
          $sformat(msg, "[%s] %s: threshold = %0d",
                        get_name(), obj.get_full_name,
                        get_threshold());
      (obj != null) && (lineno != 0) :
          $sformat(msg, "[%s] %s: threshold = %0d @ %0d",
                        get_name(), obj.get_full_name,
                        get_threshold(), lineno);
    endcase

    return msg;
  endfunction

endclass
```

References

[1] N. A. Lynch, Distributed Algorithms. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[2] UVM 1.1 Class Reference, Accellera. [Online]. Available: http: //www.accellera.org/downloads/standards/uvm

[3] UVM 1.2 Class Reference, Accellera. [Online]. Available: http: //www.accellera.org/downloads/standards/uvm