Advanced VMM Transactor Development: Tips for designing VIP you wouldn't mind reusing

Kelly D. Larson

MediaTek Wireless, Inc.
Austin, TX, USA

www.mediatek.com

ABSTRACT

VMM transactors lie at the heart of a VMM testbench, but can sometimes be tricky to implement. User guides and training material will often demonstrate basic functionality using very simple examples. This paper will attempt to delve deeper into the types of issues and problems that a verification engineer will encounter when writing a more complex VMM transactor for the types of busses he'll likely encounter in the real world. Examples will be given to illustrate possible solutions for potentially problematic areas. Tips and suggestions will also be given to help insure that the verification engineer will produce a VMM transactor that is both easy to set up and use initially, but also flexible enough to address future increasing demands from a more mature testbench environment.

Table of Contents

Table of Figures

1 Introduction

Over the last few years many companies have adopted SystemVerilog as their verification language, and are now hard at work using it to design testbenches to verify the functionality of their designs. While SystemVerilog is very capable as a testbench language, to use it to its full advantage can require the development of many lines of code. We're finding that in many cases the number of lines of SystemVerilog code developed for a block level testbench can exceed the number of lines of code for the design itself by 5 or 6 times. To justify the cost of this initial investment, it's extremely important to make sure that the code we're writing today can be reused multiple times in the future.

Many of the pieces of verification IP (VIP) that we have written for reuse take the form of VMM transactors. VMM transactors lie at the heart of a VMM testbench, but can sometimes be tricky to implement. User guides and training material will often demonstrate basic functionality using very simple examples. This paper will attempt to delve deeper into the types of issues and problems that a verification engineer will encounter when writing a more complex VMM transactor for the types of busses he'll likely encounter in the real world. Examples will be given to illustrate possible solutions for potentially problematic areas. Tips and suggestions will also be given to help insure that the verification engineer will produce a VMM transactor that is both easy to set up and use initially, but also flexible enough to address future increasing demands from a more mature testbench environment.

In Chapter 2 we'll give an overview of the VMM transactor, and take a look at using the Synopsys 'vmmgen' tool as a good starting point for code development. Chapter 3 will discuss the implementation of the VMM transactor, and weigh various design considerations and tradeoffs. Chapter 4 will go beyond basic functionality, and discuss features that are necessary to make the transactor both easy to use, and flexible enough to address future needs. Chapter 5 will contain the summary and conclusions.

2 Getting Started

2.1. VMM Transactor

2.1.1. What is it?

Transactors are an integral piece of most testbenches, and can take many forms. Anything that operates on a transaction can be implemented as a VMM transactor. In addition to typical master and slave BFMs, scoreboards, checkers, reference models and monitors can all be written as VMM transactors. Transactors can also be used to translate from one type of transaction object to another. Some busses such as AHB and AXI rely on special bus fabrics to perform arbitration and other tasks in a system. These components are also usually implemented as transactors.

While this paper will focus mainly on master and slave transactors, many of the suggestions and techniques can be applied to other types of VMM transactors as well.

2.1.2. Considerations for VIP designer

When implemented properly, VMM can enable higher productivity within design teams by promoting reuse of testbench components. Reuse doesn't happen automatically, however, as it's still possible to design VMM components that nobody ever wants to see again.

Two factors that will make people want to reuse a VMM transactor are:

1. It's easy to use.
2. It's flexible.

A VMM transactor should be easy to use. The amount of time needed to get an initial testbench up and running should be minimal. A good VMM transactor is able to provide a large amount of functionality, with very little initial investment right out of the box.
A complete environment is never developed all at once, however. Additional features are layered in as the project progresses. While a transactor needs to be easy to use for the initial testbench, it also needs to be flexible enough to address the growing demands from an increasingly mature test environment. These two factors are not mutually exclusive, and with careful planning it's possible to create components that can satisfy both.

2.2. vmmgen

vmmgen is a template generator from Synopsys, used as an aid in writing VMM components. This is an extremely useful tool, and should be your starting point when implementing a new VMM transactor, especially if you're just getting started with VMM.

```
Which template do you wish to generate?

From Standard Library:
    0) Physical interface declaration
    1) Transaction descriptor
    2) Transaction descriptor testcase
    3) Driver,          Physical-level, Half duplex
    4) Driver,          Physical-level, Full duplex
    5) Monitor,         Physical-level, Half duplex
    6) Passive Monitor, Physical-level, Full duplex
    7) Driver,          Functional-level, Half duplex
    8) Driver,          Functional-level, Full duplex
    9) Monitor,         Functional-level, Half duplex
    10) Passive Monitor, Functional-level, Full duplex
    11) RAL physical access BFM, single domain
    12) RAL physical access BFM, multiplexed domains
    13) Verification Environment
    14) RAL-based Verification Environment
    15) Testcase
Select [0-15]:
```

Figure 1 – vmmgen Main Menu

Figure 1 shows the main menu that comes up when you run the *vmmgen* tool. There are several types of transactors to choose from, and the terminology used is a little vague. 'Driver' is used to generate templates for master transactors, 'Monitor' can be used to generate templates for slave transactors, and 'Passive Monitor' is used for, not surprisingly, a passive monitor. 'Functional-level' means a transactor that deals with one type of transaction as an input, and another type of transaction as an output, usually used to translate between the two types of transactions. The more classic type of transactor is the 'physical-level' transactor, which receives transactions on one end, and wiggles signals through an interface on the other. For the master transactor, or 'Driver', the difference between the full and the

half duplex choice is whether or not it comes pre-equipped with an observation channel. The template for Half-duplex is considerably less complex if you want to start simple, you can always add additional functionality later if you wish.

Transactor templates generated with *vmmgen* can save you time by implementing the basic features which should be common to all VMM transactors. All of the code to handle the transactor configuration object is already in place, along with some basic callbacks, and reset handling. A simple transaction control loop is also implemented which might be useful for a very basic bus protocol, however it assumes an in-order atomic execution with a blocking completion model. The vmm_channel::active(), vmm_channel::start(), vmm_channel::complete() and vmm_channel::remove() methods which are used in the vmmgen template cannot be used for protocols which support multiple concurrent transactions.

3 Designing the VMM Transactor

3.1. Understand the transaction flow

Before launching into the coding of a new transactor, it's helpful to become very familiar with the underlying bus protocol. It's important to have a good understanding of everything that happens on each clock edge of a transaction. In a heavily pipelined system, there can be many things going on at the same time. On a single clock edge data might be strobed for a previous transaction, while at the same time an address is driven for a new transaction, while at the same time a request line is asserted to anticipate the arrival of a future transaction.

Up-front planning and architecture of a transactor is very important to creating reusable IP. A diagram or flow chart can be very useful here to help organize all of this information, and might also help determine concurrent activities that can be split into independent threads. Most often features are built into a transactor one piece at a time. If the overall big picture is not considered up front, you might find yourself having to rewrite previously working portions of the code when it comes time to shoehorn in a new feature.

3.2. Main control loop

At the heart of every VMM transactor is a loop to handle incoming transactions, referred to here as the main control loop. It is the responsibility of this main control loop to suspend activity if the transactor has been stopped, and to continually pull new transactions from the channel as they arrive.

3.2.1. Back to back transactions

A common challenge for verification engineers who are writing VMM transactors, even for fairly simple protocols, is to insure that unnecessary delay is not introduced between transactions. Often bus protocols allow for transactions to be issued with no delay between transactions, or very often transactions can even overlap one another.

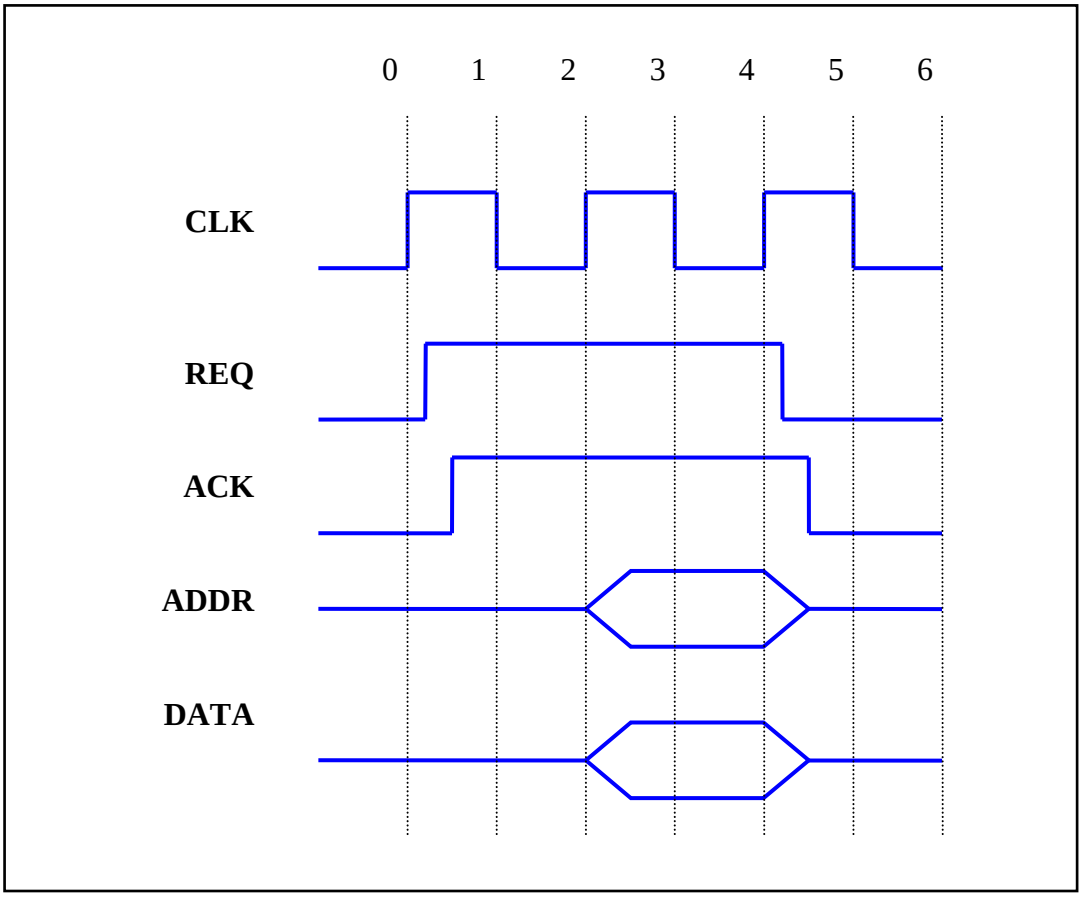

Figure 2 – Simple Bus Transaction

Figure 2 shows the waveform of a very simple bus transaction. From the bus master perspective, the address, data and request signals are all outputs, and the acknowledge signal is an input. Outputs are driven on the rising edge of the clock, and inputs are sampled on the negative edge of the same clock. Figure 3 shows the source code for a SystemVerilog interface block used in this simple example.

```
interface simple_if (input bit clk, reset);
   wire        req;   // Request
   wire        ack;   // Acknowledge
   wire [7:0]  addr;  // Address Bus
   wire [15:0] data;  // Data Bus

   clocking clk1 @(posedge clk);
     output      #1 req, addr, data;
   endclocking

   clocking clk2 @(negedge clk);
      input    #1 ack;
   endclocking

   modport SPort (
     clocking clk1,
     clocking clk2,
     input reset
     );
endinterface: simple_if
```

Figure 3 – Simple Interface Block

If the main control loop of the VMM transactor is written with only a single transaction in mind, one might easily end up with code similar to that shown in Figure 4.

```
class master_bfm;
<...>
protected virtual task main();
   simple_txn txn;
   forever begin: main_loop
      wait_if_stopped_or_empty(in_chan);
      in_chan.get(txn);
      <...> // txn processing
      @(bus.clk1);
      // Drive request
      bus.clk1.req <= 1;
      // Wait for acknowledge
      do @(bus.clk2); while (!bus.clk2.ack);
      @(bus.clk1);
      // Drive address & Data
      bus.clk1.addr <= txn.address;
      bus.clk1.data <= txn.data;
      @(bus.clk1);
      // Quiesce Bus
      bus.clk1.req <= 0;
      bus.clk1.addr <= 'z;
      bus.clk1.data <= 'z;
   end: main_loop
endtask: main
<...>
endclass: master_bfm
```

Figure 4 – Main Control Loop Simple Example #1

In this simple example, transactions are first removed from the channel, and then on the rising edge of the next clock the request line is driven high. Now the acknowledge signal is sampled on each falling edge until it's seen to be high, at which point the master will drive the address and data lines on the next rising clock edge. The transaction completes on the following rising edge, the bus is quiesced, and we return to the top of the loop for the next transaction.

The code in Figure 4 will work fine to generate a single transaction exactly as shown in Figure 2. The problem comes when we try to do back to back transactions, which will generate the waveform shown in Figure 5.

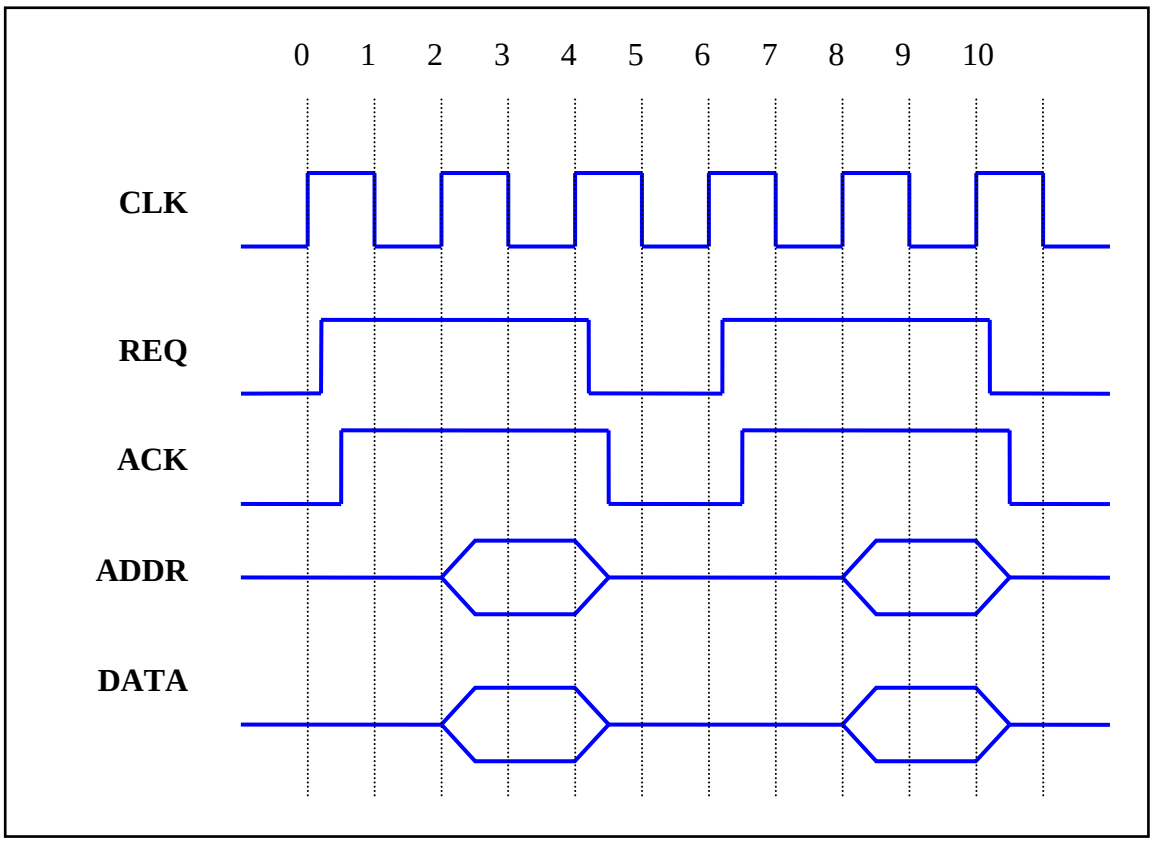

Figure 5 – Dead Cycle Between Transactions

Because of the way the main control loop was written, there will always be a dead cycle between back to back transitions. But what if this is not what we want, how can we fix it?

One way might be to 'peek' ahead before deasserting the request line at the bottom of the loop, and see if there is another transaction waiting in the channel. The problem with this method is that, as discussed later in this paper, we may decide not to actually drive the next transaction after pulling it out of the channel, and simply discard it. We can't know for sure that the transaction will make it out to the bus until after we've finished processing it.

Another method might be to set a flag at the bottom of the loop, to show that we're already driving the request line. This would allow us to skip the request step at the top of the loop. The problem with that

method is that if there isn't another transaction already waiting in the channel, or if the transactor has been stopped, we'll block at the wait_if_stopped_or_empty call, leaving the request line high.

One possible solution that offers a clean way to address this, and without any additional lines of code, is to rely on the storage characteristics of the clocking blocks themselves. Clocking blocks are a very handy feature that were added to SytemVerilog to help simply testbench development, and avoid race conditions. Signals driven by a clocking block can be assigned at any time, but the clocking block will insure that the signal is only driven onto the bus at the appropriate time. We can make use of this feature to simplify the main control loop for this example BFM.

```
class master_bfm;
<...>
protected virtual task main();
   simple_txn txn;
   forever begin: main_loop
      wait_if_stopped_or_empty(in_chan);
      in_chan.get(txn);
      <...> // txn processing
      // Drive request
      bus.clk1.req <= 1;
      @(bus.clk1);
      // Wait for acknowledge
      do @(bus.clk2); while (!bus.clk2.ack);
      @(bus.clk1);
      // Drive address & Data
      bus.clk1.addr <= txn.address;
      bus.clk1.data <= txn.data;
      @(bus.clk2); // Wait halfway through the cycle
      // Queue up possible quiescent values
      bus.clk1.req <= 0;
      bus.clk1.addr <= 'z;
      bus.clk1.data <= 'z;
   end: main_loop
endtask: main
<...>
endclass: master_bfm
```

Figure 6 – Main Control Loop Simple Example #2 (Fixed)

Figure 6 shows the new version of our simple BFM control loop. What is different is that now, after driving the address & data busses, we only wait for clk2 before writing quiescent values onto the bus. But wait, isn't clk2 the falling edge of our system clock? It is, but the clocking block will not drive any signals onto the bus at this time.

Now when we return to the top of our main control loop, one of two things will happen. If there's not a transaction ready to go we'll block, and when the next rising edge comes around all of our previously queued up quiescent values will be driven onto the bus. If on the other hand there's another transaction that's ready to go immediately, we'll fall directly through to the code which drives the request line high, overwriting the previous low value for the request line before it ever has a chance to be driven onto the bus.

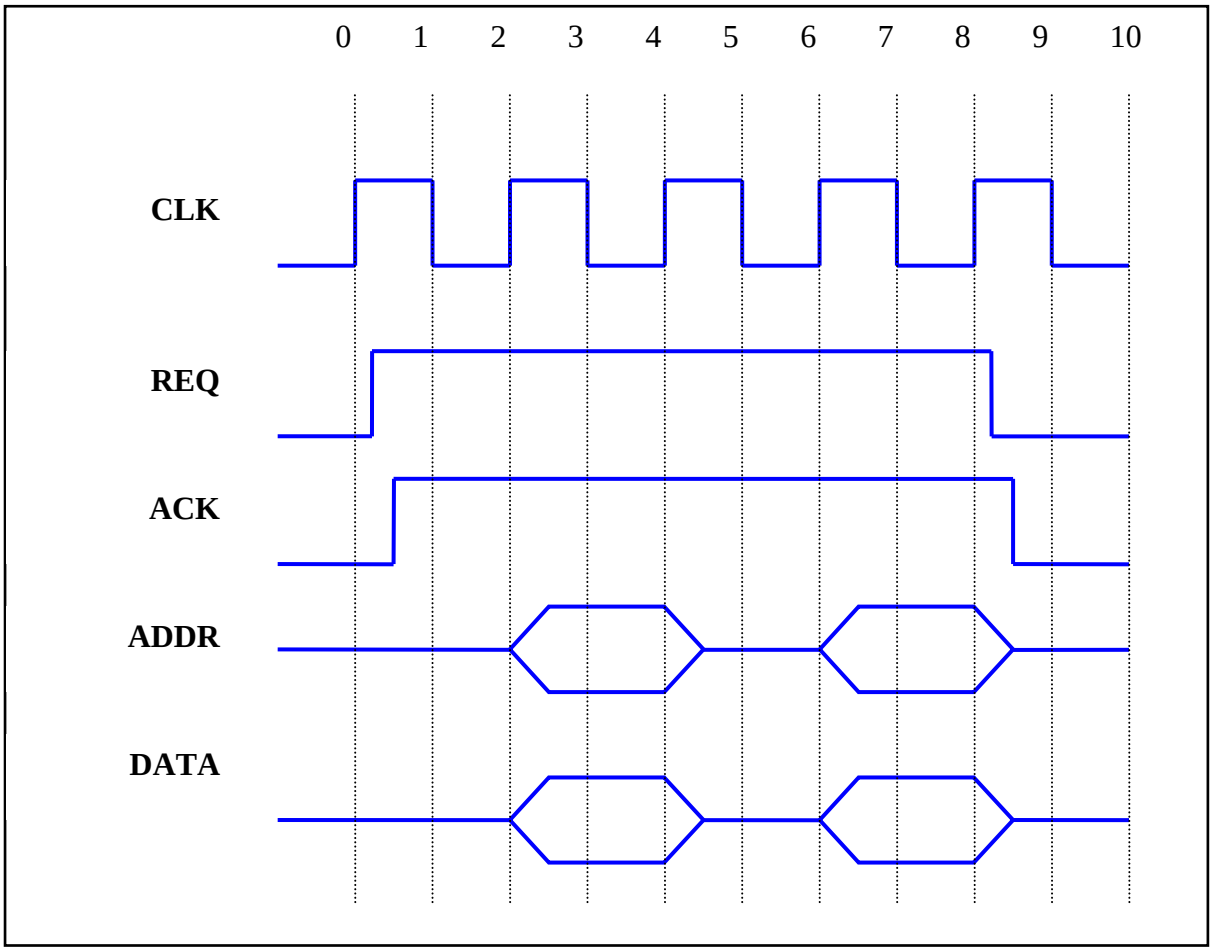

Figure 7 – Back to back transactions without dead cycle

The result of all this is that we can now drive back to back transaction without any dead cycles between, as shown in Figure 7. While using the storage characteristics of the clocking block may not work for you in all cases, many times it can help simplify the main control loop by eliminating the need to explicitly retain state information between sequential transactions.

3.2.2. Two-Thread control loop approach

Section 3.2.1 discussed various ways to drive a back to back transaction using a simple bus protocol. Most bus protocols are quite a bit more complex than that example, and are often heavily pipelined with multiple transactions happening at the same time. In such cases, it's often better to divide up the control loop into more than one thread, rather than try to implement everything inside of a single, more complex loop.

This section will show an example using a two thread approach. The first thread will take care of getting new transactions from a channel, and handling any necessary processing and delays. The second thread will own the task of actually driving the transactions onto the bus. The technique discussed in section 3.2.1 for back to back transactions can be used in this thread. Communication between these two threads can be done through a simple queuing mechanism.

Figure 8 shows a simple block diagram of the first thread. There are three main steps within this thread. The first step is to wait until there is an available transaction, and then process the new transaction through an initial callback routine. This callback routine can alter the transaction itself, perhaps injecting errors, changing data, or adjusting delay parameters. A mechanism should also be provided to signal that the transaction be dropped at this point. This is not a difficult requirement, however, since at this time the transaction hasn't yet influenced the behavior of the transactor, or any other ongoing transactions.

Once a transaction has made it past the first step, it moves into the second step. Here the transaction has made it past the "front gate," so it has been accepted by the transactor, but is still waiting to become active. While it's waiting in thread #1, thread #2 can look to see that there is a transaction pending, so it can anticipate necessary actions, such as retaining bus ownership.

As soon as thread #1 detects that thread #2 is ready for another transaction, the transaction moves on to step 3. Step 3 allows for delay to be inserted between transactions. If the transaction is configured to have any issue delay, this delay is inserted before the transaction becomes active. This issue delay is described in more detail in section 4.3.5,

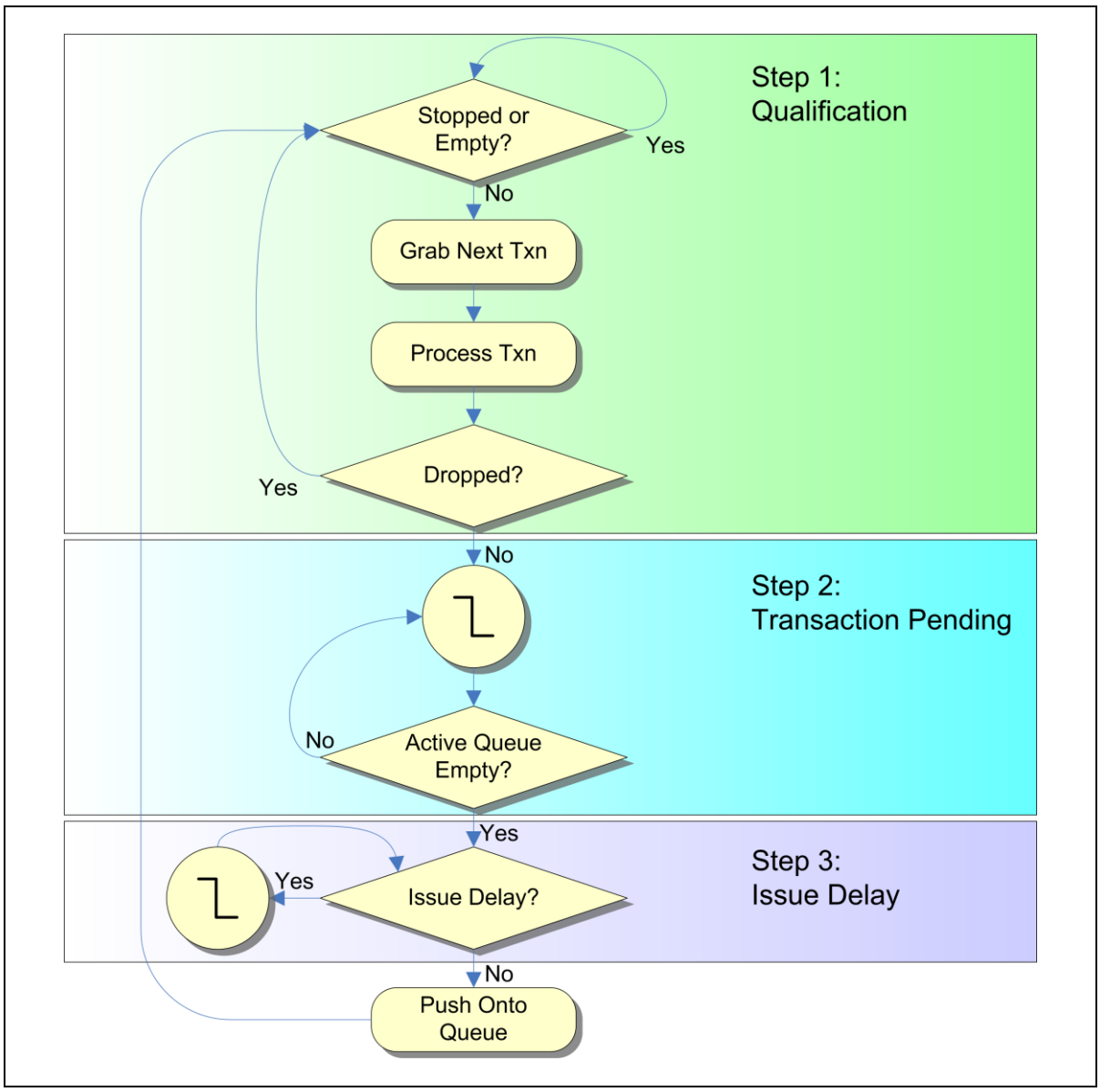

Figure 8 – Two-Thread Control Loop, Thread #1

The second control loop thread, shown in Figure 9, simply loops endlessly, driving any active transactions onto the bus. Whenever it needs a new transaction, it pops it out of the transaction queue. Although the diagram for thread #2 appears to be much simpler than thread #1, the actual implementation is usually much more involved than thread #1, depending on the bus protocol. Hardly any detail is shown in Figure 9, as this part of the transactor tends to be very implementation specific.

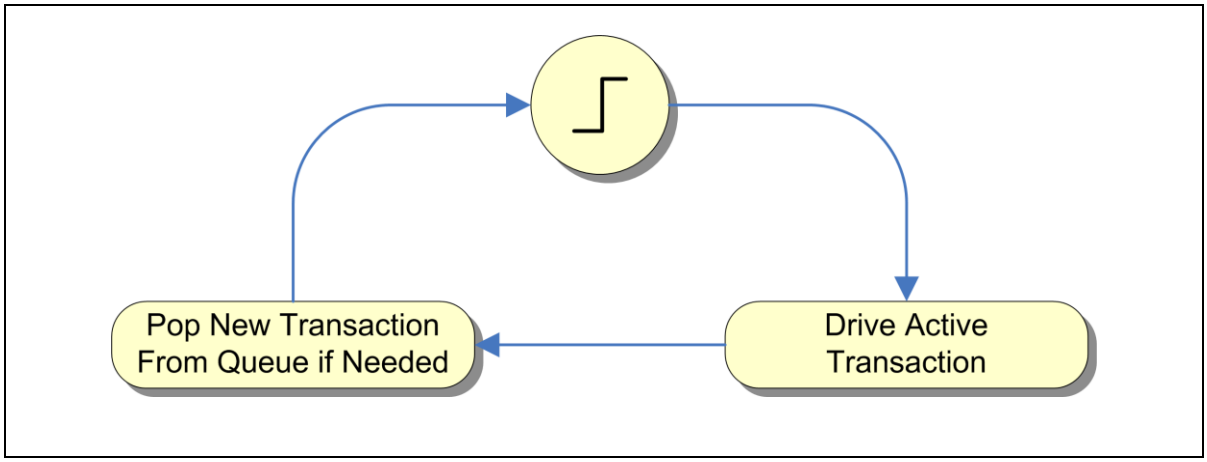

Figure 9 – Two-Thread Control Loop, Thread #2

To eliminate race conditions between thread #1 putting transactions into the transaction queue, and thread #2 taking things out of the queue, it's easiest if the threads are either synchronizing to different clock edges, or somehow delayed from one another. If a new transaction is always initiated on a rising edge, for example, thread #2 could synchronize to the rising edge, while thread #1 pulls things out of the channel and does all of it's activity on the falling edge. If this type of arrangement is not possible, other synchronization techniques should be used to insure that there is a fixed relationship between the two threads.

```
task main();
  super.main();

  fork
    drive_txn();
  join_none

  forever begin: channel_loop
    bit drop;

    // Reset pointer to next transaction
    next_txn = null;

    // Grab the next transaction
    wait_if_stopped_or_empty(in_chan);
    in_chan.get(next_txn);

    // Service pre_trans callback
    drop = 0;
    `vmm_callback(dspbus_master_callbacks,
                  pre_trans(this, next_txn, drop));

    if (drop) continue;

    // We're using negedge here to avoid race with drive_txn
    do @(dspbus.clk_mf);          // Go out half cycle
    while (txn_queue.size()>0);  // Wait for prev txn to clear

    // Insert delay issue
    if (!cfg.ignore_issue_delay)
      repeat (next_txn.delay_issue) @(dspbus.clk_mf);

    // Shove txn into queue for driving onto bus
    txn_queue.push_back(next_txn);

    // Let the world know we've started it
    next_txn.notify.indicate(vmm_data::STARTED);

   `vmm_trace(log, "Starting transaction...");
   `vmm_debug(log, next_txn.psdisplay("   "));

  end: channel_loop
endtask: main
```

Figure 10 – Multi-Thread Control Loop, Thread #1 Source

A source code example of thread #1 is shown in Figure 10. While this example uses a simple queue of transaction objects to pass transactions from thread #1 to thread #2, this doesn't have to be the case. With some bus protocols, such as AHB, it might be easier to break a larger transaction up into smaller logical chunks. For such an implementation, the queue might actually consist of objects pertaining to individual data 'beats' of the transaction.

Another advantage of using this type of queuing mechanism between the threads is for protocols where new transactions can spring into existence. As an example, an AHB wrapping burst transaction can be

terminated before completion. In this case, the AHB master may have to complete the data transfer by reformulating the original transaction into multiple smaller transactions of a different type. In this case, these newly formed transactions can be generated and inserted straight back into the transaction queue from thread #2, without affecting the operation of thread #1.

While this example shows only two separate threads, this technique can be extended to use more threads. This can be useful in more complicated protocols, such as AXI, where address, read and write channels operate somewhat independently. Breaking such a complicated protocol into multiple threads can keep the main control loop from getting too unwieldy, and thus error prone and difficult to maintain.

In some cases where independent loops are synchronized to the same clock edge, it may be necessary to enforce a set order to their execution to potential eliminate race conditions as transactions are passed between them. Without this ordering, additional delays between phases might inadvertently be introduced into a transaction. Figure 11 shows an example where semaphores are used to enforce a known order of evaluation between three concurrent loops, all synced to the same clock edge.

```
class AXISlaveBFM extends vmm_xactor;
<...>
   semaphore     write_address_channel_complete;
   semaphore     write_data_channel_complete;
<...>

function new(<...>);
<...>
   // Initialize semaphores
   write_address_channel_complete = new(0);
   write_data_channel_complete    = new(0);
<...>
endfunction: new


task write_address_channel();
<...>
   forever begin: forever_loop
      @(posedge sigs.aclk);

      <...> // Process write address channel

      write_address_channel_complete.put(1);
   end: forever_loop
endtask: write_address_channel


task write_data_channel();
<...>
   forever begin: forever_loop
      @(posedge sigs.aclk);
       write_address_channel_complete.get();

      <...> // Process write data channel

       write_data_channel_complete.put(1);
   end: forever_loop
endtask: write_data_channel


task write_response_channel();
   <...>
   forever begin: forever_loop
      @(posedge sigs.aclk);
      write_data_channel_complete.get();

      <...> // Process write response channel

   end: forever_loop
endtask: write_response_channel

endclass: AXISlaveBFM
```

Figure 11 – Forcing loop execution order

3.3. Finishing the transaction

In a complex transactor, transactions can often complete in multiple places scattered throughout the code. Often read and write transactions are handled in multiple threads, and there are sometimes a variety of reasons for a transaction to complete in non-normal ways, such as an error conditions, or loss of bus ownership. Good programming practice, and a desire to keep one's sanity, dictates that all transactions should complete through a common routine.

This "transaction finished" routine is an ideal place to put all of the common bookkeeping and logging necessary for a VMM compliant transactor. The final callback is the most important part of that routine, since this could potentially be used by the testbench for measuring coverage, or for feeding a scoreboard. While this is an obvious necessity for a monitor transactor, it's important for all other types of transactors as well.

```
function void transaction_finished(dspbus_txn txn);
  // Final Callback
  `vmm_callback(dspbus_master_callbacks,
                post_trans(this, txn));

  // Transaction Tracing
  `vmm_trace(log, "Completed transaction...");
  `vmm_debug(log, txn.psdisplay("   "));

  // Indicate that the transaction is finished
  txn.notify.indicate(vmm_data::ENDED);

  // Observe transaction
  if (obs_chan != null) obs_chan.sneak(txn);

endfunction: transaction_finished
```

Figure 12 – Finishing up the transaction

This is also a good place to use the notification object of the VMM transaction object to indicate that the transaction has ended, and to call typical VMM log trace and debug messages. In a heavily pipelined environment, the 'vmm_data::ENDED' transaction notification is often the best method that a testbench environment has for knowing exactly when a transaction has finished.

Another optional feature would be to take the completed transaction, and 'sneak' it into an observation channel, if one was provided when the transactor was instantiated. Again, this is a necessity for a monitor transactor, but it could also be useful for a master or slave transactor. Although it is usually preferable to use a monitor transactor for coverage or scoreboarding, providing an observation channel with your master and slave transactors could ease the burden of the verification engineer putting together a quick initial testbench. This could be considered optional, because the final callback can provide the same information as an observation channel, but the implementation only takes a couple lines of code, and some people might prefer using channels over callbacks.

3.4. Slave Considerations

3.4.1. Memory modeling

For most applications it is useful for the slave transactor to provide for a way to model memory. In this way if a value is written to a particular address, you can expect to be able to read that same value from the same address. There are three common methods for doing this:

1. Internal Memory Model
2. Callback routine
3. Response Channel

For method 1, the response data is provided by the slave, while for methods 2 & 3, response data is provided by the testbench. These methods are not mutually exclusive, the transactor can provide the user with the option of using the method they prefer.

3.4.1.1. Internal Memory Model

For a system with fixed width transactions, memory can be easily modeled using a simple SystemVerilog associative array. Many protocols allow varying of transaction widths, however, so it's more likely to be a bit more complicated than that, and implementing the memory model in a class object is probably the way to go. The memory model class should provide access functions which allow the user to seed their own values, and to query existing values for back door type accesses. Modeling memory in a class has the additional benefit of making it easy for a user to extend the class to provide custom behavior, such as changing the behavior when reading from uninitialized memory locations.

The memory model object within the slave transactor should either be declared as public, or the slave should provide access functions to allow this object to be changed, as many systems have the need for multiple transactors to share the same global memory space. If multiple transactors can be configured to reference the same memory object, they can easily simulate this. If this capability is not provided, you might force additional complexity into the top-level testbench.

3.4.1.2. Callback Routines

Callback routines can also be used to determine what data values the transactor will return. The transactor should already be written with callback routines which are called just before data is written to, or just after data is read from the bus. Since the transaction object is passed to the testbench during the callback, the verification engineer could use the callbacks to override the behavior from the built in memory model if desired.

Care should be taken to make sure that the internal memory model accesses, and the callback routines, are called in the correct order. For reads, the internal memory model routines should update the transaction object first, then the callback should called, and then the data should be driven onto the bus. With this order the callback will have access to the data values from the internal memory, and also be able to override if it wants.

3.4.1.3. Response Channel

Transaction data can also be obtained by communication with the testbench via a response channel. This method can be more complicated to implement than a callback, since you may have to take precautions to insure that the interaction through the response channel doesn't consume time, and prevent the slave from functioning correctly. This concern can be more easily managed with a callback routine

simply by implementing the callback as a function, rather than a task, which insures that it is executed in zero time. See [3] for more information on how to properly implement a response channel.

Having internal memory models implemented within a slave transactor helps an engineer get an initial testbench up and running fast. A mechanism should also be provided, however, for more sophisticated control which may require direct interaction by the testbench environment. If callback routines are implemented correctly, they should be sufficient for this type of control. Response channels can also be implemented, especially if it's not known exactly how the transactor will be used. If the use model is understood, you may consider treating response channels as optional, since they're generally more difficult to use than callbacks.

3.4.2. Active/Passive mode or separate monitor?

The job of a passive monitor is to observe bus signals, and translate them into a more abstract VMM data object. Many times the logic for a passive monitor is very similar to that of a slave transactor, with the only difference being that the monitor observes slave response signals, rather than actively driving them. Because of the similarity in the logic, it might be attractive to make a single transactor that could be configured to do either job, rather than having a separate implementation for both the slave and the monitor. Having a single implementation may be better with regards to code reuse and maintenance.

We still tend to favor having separate implementations. One reason for this is that the passive monitor needs all signals to be inputs, where the slave needs some of the signals as outputs. Since modports enforce directionality on the signals, you would either need to instantiate the slave transactor with both sets of modports, or use one modport and declare all actively driven signals as 'inout'.

3.4.3. Use factory patterns for new transactions

When a slave or monitor observes the beginning of a new transaction, it automatically creates a new transaction object. The exact transaction object which is created should always be a user-overridable factory template. It's impossible to know what the future unpredictable needs of users are for annotating transactions with arbitrary information, and this allows the use of any extended transaction object. More information on how to do this can be found in [3].

4 Enhancing the VMM Transactor

Chapter 3 discussed issues and concerns with getting a basic transactor implementation up and running. This chapter will discuss additional enhancements, once the basic functionality is in place.

4.1. Callbacks

It's impossible for an engineer developing a VMM transactor to anticipate all the possible needs that a testbench environment will require. If a transactor doesn't provide the ability to address a particular testbench need, a verification engineer may end up with no other choice but to start hacking on the existing code in an attempt to get the functionality, slowing down the schedule and introducing additional risk from unintended errors. Callbacks are the best, cleanest way to insure that the transactor will have the flexibility needed to adapt to unknown future requirements.

Callbacks are simply hooks placed in strategic points in the operation of a transactor, and have many uses. Callbacks can be used to inject errors, measure coverage, monitor performance, model system memory, check data, set timing delays, the list goes on and on.

Callbacks are extremely easy to implement, as they're basically empty subroutines. At the very minimum every transactor should have a callback placed immediately after transactions are taken out of the channel, and another callback placed at the completion of a transaction as a final step. The *vmmgen* tool generates an initial implementation for both of these callbacks for you, so there's really no excuse for not having both of these. Another obvious place to put callbacks is at the beginning and end of each beat of data for bus protocols which support burst transfers. Any place within the main control loop where transactions can be aborted, or error conditions can exist would be another good candidate.

Callbacks can be implemented as either tasks or functions. Tasks are more flexible, and should be used whenever timing isn't an issue. Tasks can block, however, and so most of the time callbacks will probably need to be implemented as functions, to insure that they don't delay timing critical behavior and interfere with the normal operation of the transactor.

4.2. Notifications

Notifications are used primarily to help synchronize events in the transactor with the testbench. This is another area that can help make the transactor easier to work with. All VMM transactors already contain a vmm_notify object in their parent object, as do all vmm_data objects.

4.2.1. vmm_data notifications

At the very minimum, the transactor should indicate the vmm_data::STARTED and vmm_data::ENDED notification on all transaction objects at the appropriate places. In a heavily pipelined system, this is often the best way that the testbench has of synchronizing transactions with other external events. These notifications have been discussed elsewhere in this paper, see Figure 10 and Figure 12 for code examples of this.

4.2.2. Transactor notifications

Notifications can also be used to indicate important events to the testbench. This could potentially help measure coverage, and track interesting corner cases. Often times systems will implement a subset of a particular bus protocol. While some bus activity might be completely legal from a protocol sense, such as early aborted burst transactions, unaligned data accesses, or lock transactions, they may not be supported within the design itself. If a transactor is written to indicate notifications for these types of events, checkers can easily be added to the testbench to insure that the prohibited activity does not occur within the design.

Other interesting events, such as a master losing the bus grant, or an arbiter changing the bus owner should also be indicated through notifications.

4.2.3. Notifications and Consensus

During the debug process, it can be a very common occurrence for a test to end with a testbench timeout. This usually means that something has locked up somewhere, and prevented forward progress. Using a vmm_consensus object in the main VMM environment is a great way to get better information on why a test failed, and speed up the debug process. Each transactor in the system can be easily registered with this consensus object by using a single vmm_consensus::register_xactor() call, but only if the transactor has been written to support this. Adding this type of support is not difficult. A transactor

simply needs to indicate the vmm_xactor::IS_BUSY notification while it has active transactions, and vmm_xactor::IS_IDLE when it is idle. If it's written to do this, and forward progress in the test stops because the transactor is waiting for a response that never comes, then after the testbench times out the vmm_consensus object can provide information on which transactors still have pending transactions.

This IS_BUSY/IS_IDLE notification methodology can be expanded upon to provide even more detailed information about what the transactor was doing when forward progress halted. Notifications could be written to show that it was waiting for an address grant which never arrived, or separate notifications for read and write data response from the slave. See [5] for code examples showing this type of usage of the vmm_consensus object.

4.3. Mixing things up with random timings

4.3.1. Why important?

When making a good VMM transactor, it's not good enough to simply generate a stream of legal transactions, the transactor has to be capable of generating ANY legal transaction. If something can be randomized, if a signal can be shifted, if other signals truly are "don't care" for a period of time, then given enough time, the transactor should be capable of hitting all these variations.

One common mistake when writing a transactor, however, is to assume that if the transactor can generate valid transactions with random timings, then it's sufficient to simply randomize a transaction within a loop, or use an atomic generator, and we're good. In addition to random timings and delays within the transactions themselves, we also have to pay attention to randomizing the timing *between* transactions.

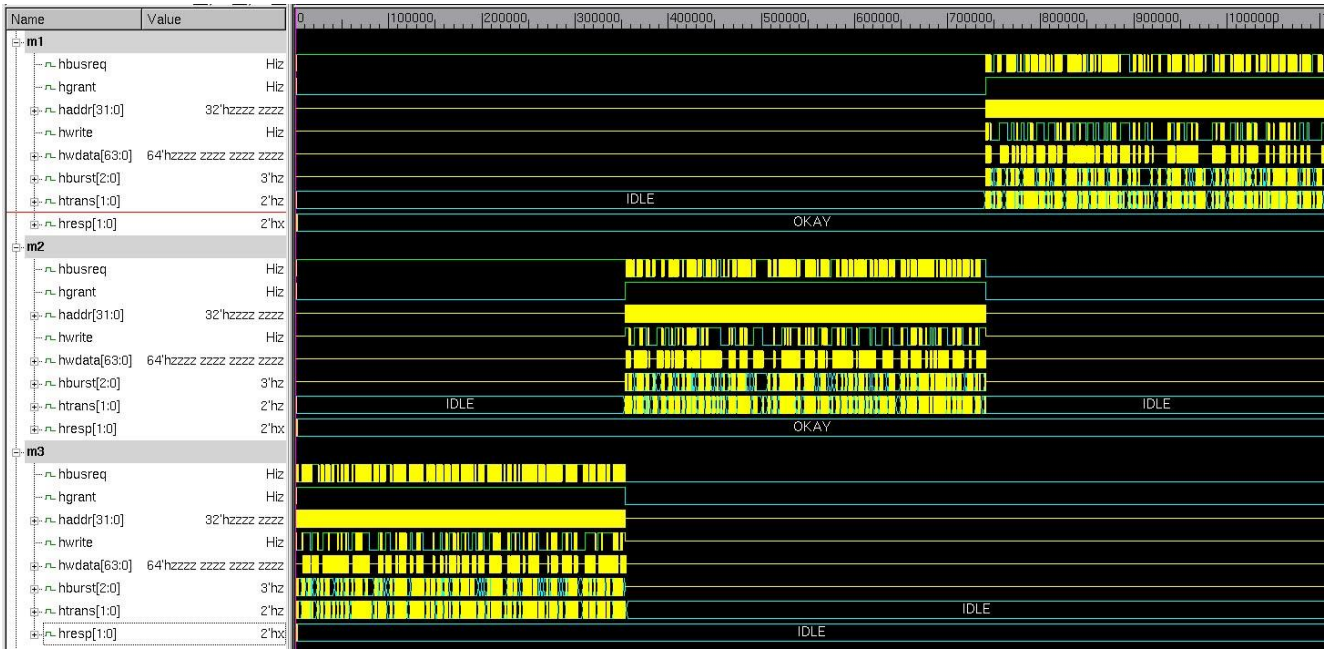

Figure 13 – No delay between transactions

Figure 13 shows waveforms from a system where there are three master transactors attached to a simple arbitration block. The transactions themselves are completely random, with random variations in timings wherever possible.

It's easy to see which bus has the highest priority, as it completely chokes out all activity on the other two busses until it's finished with all of its transactions. The second highest priority bus then takes over, and completes its transactions, followed by the third. On the plus side, this waveform shows that we've accomplished the goal of being able to drive back to back transactions, as discussed in section 3.2.1. If it was impossible to create this waveform, that would also be a problem. What's happening here, however, is that in spite of randomizing everything inside of a transaction, we haven't randomized the spacing between the transactions. This makes for an interesting testcase, but it's certainly not sufficient for fully testing the arbiter or other components in the system.

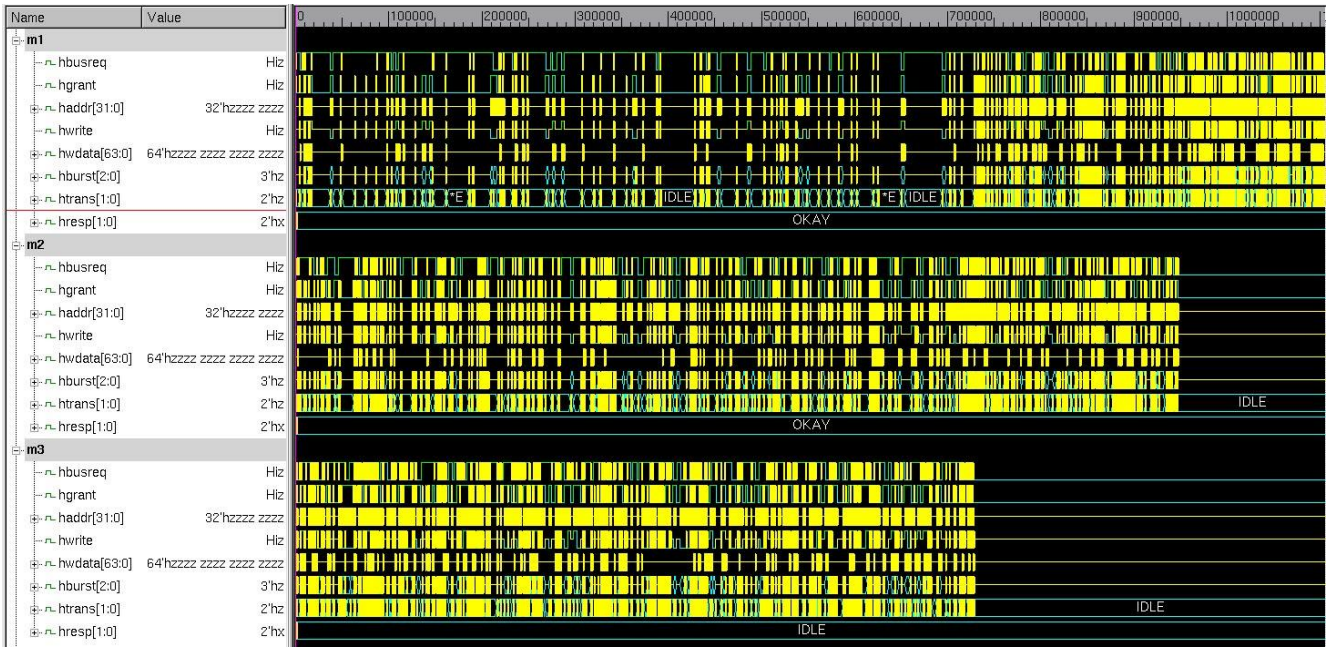

Figure 14 – Multiple busses with random issue delay

By adding a simple "issue_delay" parameter to the transaction object, we can then randomize the delay between transactions. Running the same test with this capability, we now generate the waveforms seen in Figure 14.

Adding delay between the transactions allows lower priority busses to sneak transactions through, and provides much more interesting and realistic bus behavior. Delay between transactions could also be handled in the testbench, but the transactor should supply a simple mechanism such as this to enable fast development and ease of use. Requiring the testbench to space the transactions itself would add complexity to every scenario generator written for the transactor.

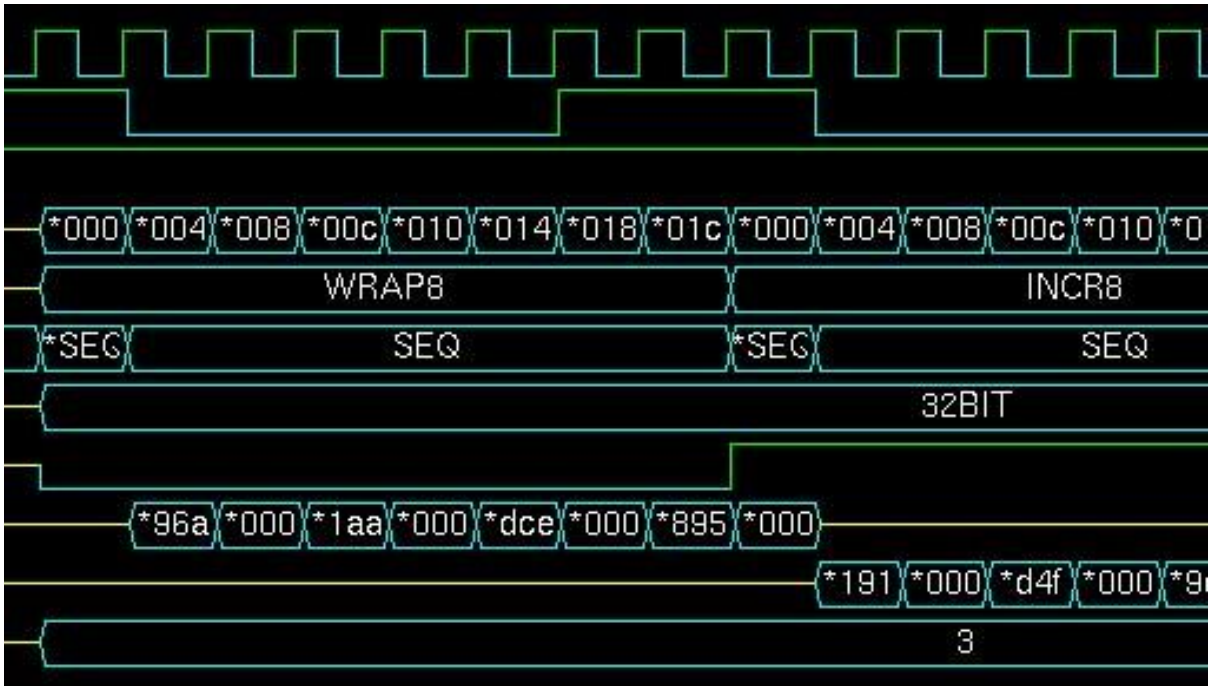

Figure 15 – Back to back transactions, issue_delay = 0

Figure 16 – Back to back transactions, issue_delay = 3

Figure 15 shows the waveform of an individual transaction with its issue_delay parameter set to 0. Figure 16 shows the same transaction with issue_delay set to 3. The code to implement this delay is very simple, and can be seen in Figure 10.

4.3.2. Types of delay

A transactor should be able to vary any aspect of a transaction that is variable. When implementing a transactor, the bus spec should be examined carefully for all possibilities. If a request line doesn't need to be held through the duration of a burst transaction, it should be dropped at any random legal point. If a master can get a grant on the same cycle it requests, a mechanism to create that condition must be implemented. Wait states, busy states, any type of delay, and anything that can be varied, must be considered.

4.3.3. Randomizing delay through transactor configuration

As previously mentioned, an engineer implementing a transactor has multiple goals. On the one hand, he needs to make a transactor easy to operate, and something that can enable an initial testbench development to proceed quickly. On the other hand, as a testbench matures, the testbench needs to be able to adapt to more sophisticated control.

Delay timings are a good example of where both of these needs can be served at once. It might be a good strategy to let the transactor be capable of determining its own random timing and delays in a fairly simplistic way, but at the same time provide the necessary hooks to enable more sophisticated control at a later date.

Let's use the example where a slave transactor needs to respond to a burst transaction from a master, one beat at a time. The slave can either respond with the data immediately, or it can use the bus protocol to insert wait states until it's ready to deliver the data.

An extremely simple implementation would be to have a configuration parameter for the slave which sets the 'latency' of the slave device for all accesses. If the value is set to 2 cycles, then the slave will delay 2 cycles on every access. This is probably a bit too simple to be of much use, a better way might be to provide a 'min' and a 'max' value for the latency, and let the slave pick something randomly within the range. This provides a nice default random behavior, but also allows the user to set a fixed value by specifying the same value for min and max.

What if the user now wants to simulate a system where the first beat of an access has a very long latency, followed by beats with much shorter accesses? Parameters could be added to the slave configuration which differentiate between the first access and subsequent accesses, but it's really not possible to anticipate all of the potential scenarios a user will come up with. Probably at this point it's best to simply provide a hook which allows the testbench to implement more sophisticated behavior through callbacks.

4.3.4. Randomizing delay through callbacks

Callbacks should be used in a transactor at any point where timing needs to be determined.

```
function int AHBSlaveBFM::StartOfBeat(AHB_txn curtrans, int curbeat);
   int hready_delay;

   // Calculate default beat delay
   if (curtrans.trans_type == AHB_txn::READ)
     hready_delay = $urandom_range(cfg.min_rd_delay,cfg.max_rd_delay);
   else
     hready_delay = $urandom_range(cfg.min_wr_delay,cfg.max_wr_delay);

   // pre_beat callback
   `vmm_callback(AHBSlaveBFM_callbacks,
     pre_beat(this,curtrans,curbeat,hready_delay));

   return(hready_delay);

endfunction: StartOfBeat
```

Figure 17 – Randomizing delay example

Figure 17 shows a sample of a slave transactor function which is called at the beginning of every data beat. First a default random delay time is calculated, as described in the previous section, and then a callback is called with this delay as one of the parameters. This allows the callback routine to override the delay value if it chooses. Using these two methods together provides a simple default way to handle random delays, but also provide the necessary hooks for more sophisticated control should the testbench environment require it.

4.3.5. Randomizing delay through transaction class data members

Delay timings can also be handled by adding additional parameters to the transaction object itself, rather than by passing values explicitly in the callback routine as in the above example. The advantage of doing it this way is that delay values could be determined through a SystemVerilog language constraint on the transaction object. This allows for much more flexibility than a simple min/max randomization as shown above, at the cost of additional complexity to the transaction object itself.

Transaction generators could then randomize these delay values along with the rest of the transaction before passing it to the master transactor. Delays for slave transactors could be calculated in a callback routine by simply calling .randomize() on the transaction object, making sure to set the rand_mode to 0 on data members which shouldn't be randomized.

4.4. Additional Features

4.4.1. "ignore delay issue" configuration flag

Another easy to implement feature that we've found useful is to add a flag to the configuration of a master transactor to ignore the delay_issue parameter of all transactions. *delay_issue* is a parameter which is used to add spacing between transactions, as described in section 4.3.1.

This configuration flag can be useful in environments where the testbench wants to completely control delays between transactions, but we've found it also works to easily add another mode of behavior into the testbench without interfering with any existing transaction generation.

By occasionally setting this flag in one or more transactors, suddenly the transactor starts saturating the bus that it's attached to, and behaving "badly" from a system perspective. In a system where you might

already have dozens of carefully crafted VMM scenarios running, this is a way of mixing up the existing behavior in a random way with very little additional effort.

4.4.2. "drive z between transactions" configuration flag

Debugging a complex system can be difficult, and a good transactor will help simplify the task wherever it can. One way it can do this is to provide detailed and clear error and logging messages. Another way is to provide clear waveform dumps. Waveform dumps can be made much easier to read by only driving values when it's legal to do so, and driving a high-impedance 'z' at all other times.

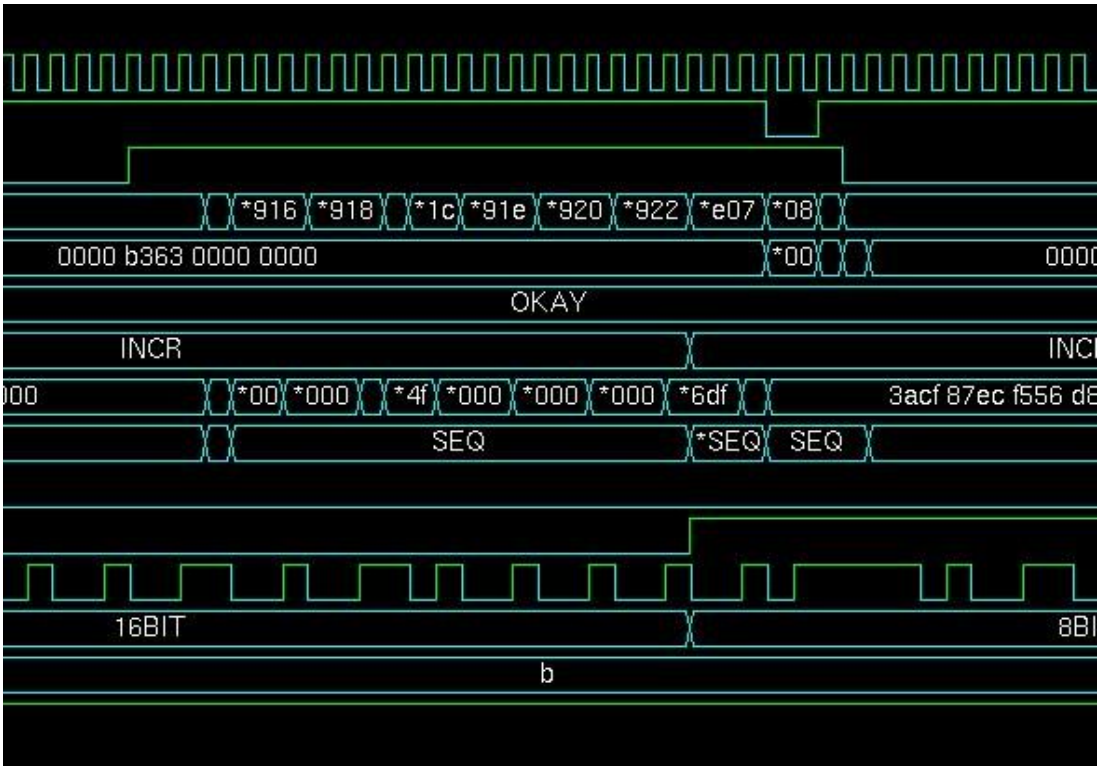

Figure 18 – Typical AHB transaction without driving Z

Figure 18 shows the waveform of a typical AHB transaction without the feature enabled to drive Z between transactions. In this mode, signal values from previous transactions are simply left on the bus, much like a typical RTL component would do.

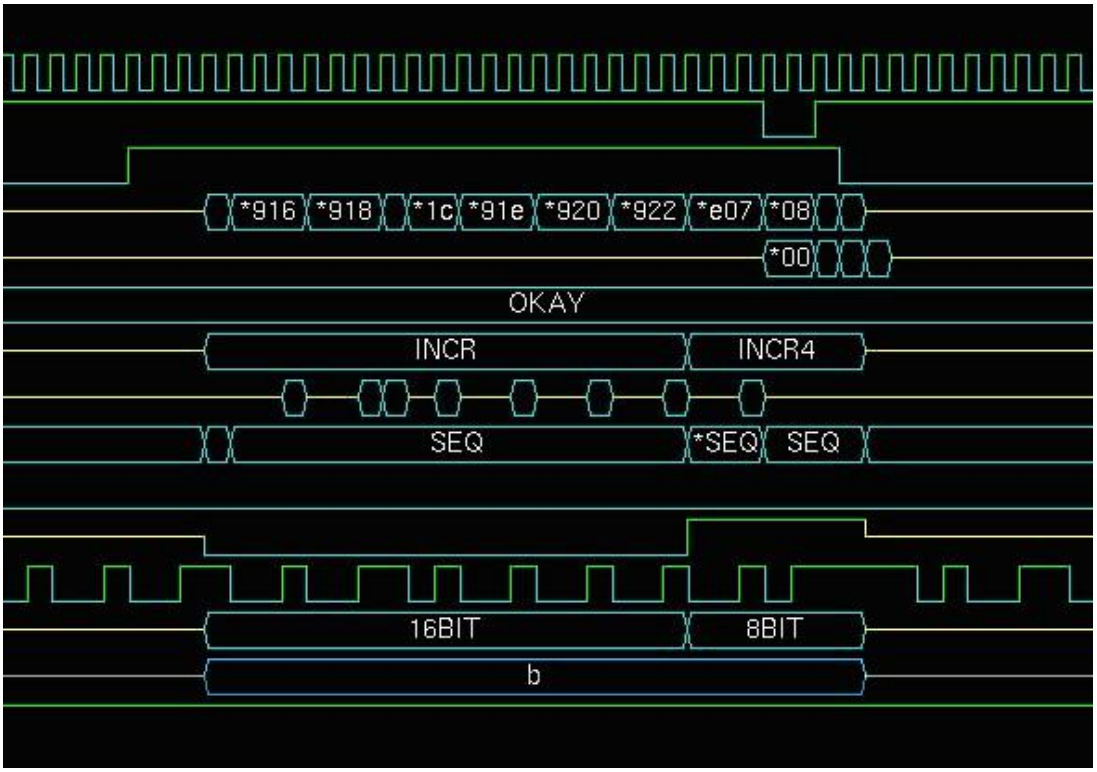

Figure 19 – AHB transaction, with driving Z enabled

Figure 19 shows the same transaction as in Figure 18, but with a configuration parameter to drive z between transactions enabled for both the master and the slave. With this feature enabled, it's a lot easer to see the individual bus transactions and data beats.

This is not just an esthetic enhancement, there's a practical side to this as well. By driving Z's whenever the bus is not being actively driven, you can potentially catch somebody latching values off of the bus when it's not legal to do so. By leaving old values on the bus, some types of illegal behavior could be inadvertently masked. If for some reason it is not desirable to have Z's on the system busses, values should at least be set to random values when they are not being actively driven, rather than leaving the old values in place.

4.4.3. Good, descriptive self reporting

Ideally, every class object should have a routine which produces as output a good description of its current status. Most engineers won't neglect the implementation of the psdisplay() function for their vmm_data class, but tend to not be as disciplined when it comes to other objects, such as transactors. Each transactor should be able to generate text which, as a minimum, reports the value of all its configuration variables. Current status, simple activity statistics, and the current size of any active queues can also be helpful. This type of routine can be a real time-saver when it becomes necessary to debug potential issues in the testbench environment itself. Inserting helpful messages with the `vmm_debug logging macro can also have a big pay off.

5 Conclusions

Developing verification IP for internal use can be a lot of work, and can consume much of the DV resources for a project. This initial investment can pay big dividends if the VIP which is developed can then be reused on future projects, giving the verification engineer more time to address those issues which are unique to his immediate verification task. In order for something to be reused effectively, it has to meet the following criteria:

1. It has to be easy to use. Ideally it requires very little effort to establish a basic functionality. This means that the VIP should come ready to go 'out of the box' with very little initial investment.
2. It has to be flexible enough to meet both foreseen and unforeseen requirements that a testbench will require as more functionality and capability is built into the environment.

This paper has tried to offer tips and guidelines for the design of VMM transactors which will address both of these criteria, and enable the design of VIP that does what you want it to do today, and that you won't mind reusing tomorrow.

6 References

[1] “Verification Methodology Manual for SystemVerilog,” Janick Bergeron, Eduard Cerny, Alan Hunter, Andrew Nightingale, 2005.

[2] “VMM Primer: Writing Command-Layer Master Transactors,” Janick Bergeron, VMM Central Web Site, http://www.vmmcentral.org/pdfs/wrtg_commd_layer_slave.pdf, 2006.

[3] “VMM Primer: Writing Command-Layer Slave Transactors,” Janick Bergeron, VMM Central Web Site, http://www.vmmcentral.org/pdfs/wrtg_commd_layer_transactors.pdf, 2007.

[4] IEEE Std 1800, “IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language,” 2005.

[5] “Using Verification IP and VMM Applications to Jumpstart Verification of an AXI Subsystem,” John Dickol, SNUG San Jose 2008