Verification Methodology Manual

SystemVerilog

Self-Paced Tutorial

Table of Contents

1 Introduction

This tutorial is a beginner's guide to using the VMM methodology, with the SystemVerilog language. You can simulate your testbenches with VCS. With using the VMM methodology, you can quickly build a layered testbench. These testbenches support high-level tests using constrained random stimulus and functional coverage to indicate which areas of the design you have checked.

In this tutorial, you will verify an ATM switch, starting with simple stimuli and then working up to a complex testbench.

1.1 Requirements

1.1.1 Knowledge

- You should know the Verilog hardware design language.
- You should have basic knowledge of the SystemVerilog language, especially concepts such as Object Oriented Programming (OOP) and synchronization between the testbench and Design Under Test (DUT). If you have not used virtual methods in OOP, you should first read Appendix A. The term "method" refers to tasks and functions.
- You should be familiar with VCS to compile and simulate designs plus testbenches.

1.1.2 Tools

This tutorial requires VCS X-2005.06 or higher.

1.2 References

You should refer to documentation and examples contained in the releases of the tools used. The Verification Methodology Manual (VMM) for SystemVerilog is a useful reference. You can buy it from Springer Science + Media or use the on-line version at **`$VCS_HOME/doc/UserGuide/vmm_sv.pdf.`** More information on the VMM book is available at `www.vmm-sv.org`.

Just to clarify, the methodology is known as the VMM methodology, while the book is the VMM book. Synopsys uses the "**`vmm_`**" prefix for the classes and macros in SystemVerilog.

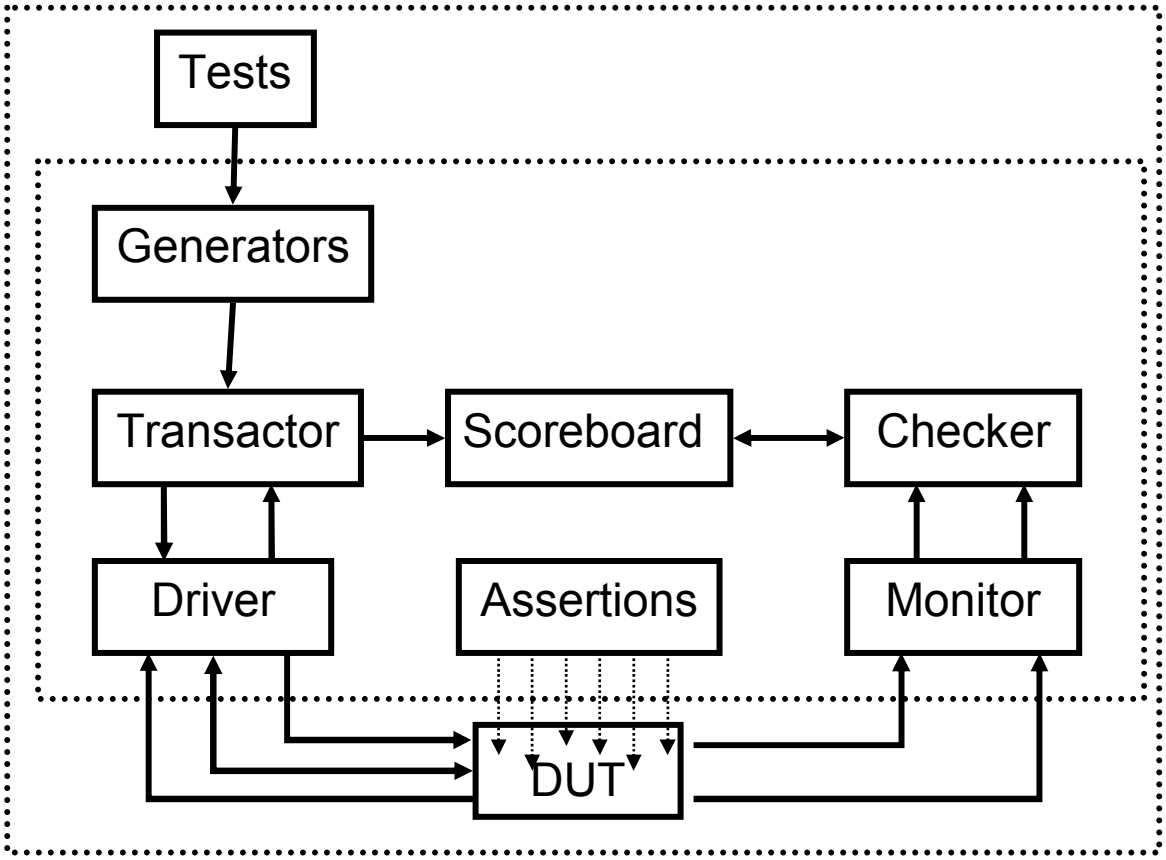
1.3 Overview

To get the most out of a Hardware Verification Language (HVL) such as SystemVerilog, you need to adopt a new methodology. If you use the same techniques from your old Verilog testbenches (directed tests with little randomization), you will not find bugs in your design as quickly as if you tried a new approach. Switching will also help make your code easier to maintain and reuse.

The VMM methodology consists of coding guidelines and a set of base classes that allow you to develop reusable testbench components such as data models, transactors, and generators. Synopsys designed VMM based on years of experience verifying many different types of designs, enabling use of advanced techniques. Using VMM will give your project a consistent look and feel, so less time is spent creating verification infrastructure, and more time verifying your design.

The VMM involves creating a robust, flexible testbench infrastructure once, and then creating many simple tests, while never changing the underlying testbench. This requires advanced techniques such as factory patterns and callbacks. For example, virtual methods allow you to write the testbench once, and then not have to modify it for every possible type of stimulus variant such as error injection, synchronization, and variable delays. While you can verify a design using a simple testbench, you would have to create many elaborate tests and continually update the testbench. This latter approach yields more code and reduces the readability and maintainability of your code.

1.4 Layered Environment



The three major parts of a verification environment are the DUT, the testbench (the inner box above), and the test which controls the testbench. Inside the testbench are the following parts:

- The Driver controls the signals into the DUT. You can write it in SystemVerilog or Verilog. It executes single commands such as a bus read or write, or driving a cell / packet / frame into the DUT

- The Monitor bundles signal changes from the DUT into transactions.
- Assertions constantly check the DUT for correctness. These look at external and internal signals. The testbench uses the results of these assertions to see if the DUT responded correctly.
- The Transactor takes high-level transactions such as a burst-read into individual read commands, or a single USB transaction into multiple USB packet TX/RX commands.
- The Scoreboard stores the transactions from the Transactor for later comparison.
- The Checker compares the output of the DUT, as seen by the Monitor, to the Scoreboard, using a predictor or reference model.
- The Generator generates transactions, either individually or in streams.

Note that each of these components may be instantiated several times or come in several flavors for different protocols.

A goal of VMM is that the testbench will not need to change for individual tests. Its components include hooks to allow the testbench to control the stimulus, without having to anticipate all possible conditions such as error injection.

1.5 Labs

Each lab exists in a separate directory, allowing you to complete the labs in any order. Each lab has some existing code and comments indicating where you need to complete code, as shown below:

```
// Lab1 - ... comments
```

Solutions to all labs are in the solutions directory. You should examine the tips and hints sections that follow each lab before consulting the solutions.

During the labs, you will construct a verification environment to verify a very simple APB system. The testbench issues read/write commands, and the RTL is a simple memory. The APB protocol uses a simple address, write-data, read-data, Read/Write, Select, Enable and clock interface.

Lab instructions are provided in Appendix B of this document.

2 Messaging

2.1 Overview

A testbench produces messages of many types and severities. The vmm_log class lets you control which messages are displayed, what their format is, and even promote and demote them (useful for error testing). All messages are sent to standard output, i.e. displayed on the screen and sent to the simulation log file, just like **`$display`**.

2.2 Type and Severity

In VMM, every message has a type and a severity. You may want to print a message to debug a piece of code, tell the user that simulation reached a notable state, or encountered a problem. The message type tells which of these is happening.

- Failure: Error has been detected
- Note: Simulation progress
- Debug: Optional simulation diagnostics
- Timing: Timing check or error

The severity field describes the message importance. The following list shows the severity levels and the type in parenthesis.

- Fatal: Functional correctness definitely compromised *(Failure)*
  - Example: Testbench failure
- Error: Functional correctness may be compromised (*Failure)*
  - Example: Actual model results don't match expected results
- Warning: Functional correctness not compromised (*Failure* or *Timing)*
- Normal: Regular, expected message
- Trace: High-level simulation execution trace message (*Debug)*
  - Example: "*Executing transaction*"
- Debug: Detailed simulation execution trace message (*Debug)*
  - Example: "*Waiting for acknowledge*"
- Verbose: Very detailed simulation execution trace message (*Debug)*
  - Example: "*Sending byte #5 (0x5A)*"

2.3 vmm_log class

Each part of the testbench (test, generator, checker, etc.) uses its own instance of the vmm_log class to generate messages. Each instance is a separate message source with a descriptive name and an instance name. You can use regular expressions to select and control sources so use clear names. Usually the descriptive name is the name of the class instantiating vmm_log, and the instance name is the name of the object, or "class" if there is only a single instance.

2.4 Declaration and Instantiation

The vmm_log is usually instantiated inside a testbench object such as a generator or checker, or in a data object:

```
vmm_log log;
```



```
log = new("name", "instance");
```

The ***name*** string is the name of the class that contains the log, such as "USB Host", or "MAC Frame". The ***instance*** string is the name of this instance of the object such as "Generator 1", or "Left side". If there is only a single instance, just use the string "class".

The easiest way to use a vmm_log object is with the macros:

```
`vmm_fatal(vmm_log log, string msg);
`vmm_error(vmm_log log, string msg);
`vmm_warning(vmm_log log, string msg);
`vmm_note(vmm_log log, string msg);
`vmm_trace(vmm_log log, string msg);
`vmm_debug(vmm_log log, string msg);
`vmm_verbose(vmm_log log, string msg);
```

Here are two examples of using the above messages. The first displays a simple string. The second needs to print variable arguments, so it uses $psprintf, which returns a formatted string:

```
`vmm_verbose(log, "Checking rcvd byte");
if (byte != expect) begin
   `vmm_error(log, $psprintf("Bad data: 0x%h vs. 0x%h",
                             byte, expect));
end
```

Note that these macros expand to several lines, so surround them with begin-end when used in an if-statement.

Coding Guideline:
Avoid declaring and instantiating an object all on one line. You will not be able to call any procedural code before the first call to **new()**. Instead of:

```
vmm_log log = new("name", "instance");  // Poor code
```

Use:

```
vmm_log log;                     // Separate declaration
log = new("name", "instance"); // from instantiation
```

Note that this tutorial occasionally skips this rule to make the examples more readable.

2.5 Message Handling

The messaging class handles each message according to its severity level. The default is that fatal messages cause the simulation to exit, error messages increment a global error count, and cause the simulation to exit after 10 errors, while all others just print to standard out. You can use the method **vmm_log::modify()** to change how messages are handled.

2.6 Controlling Verbosity

By default, only messages with a severity of NORMAL (vmm_fatal, vmm_error, vmm_warning, vmm_note) or higher are displayed. You can control this two ways:

2.6.1 Using +vmm_log_default

The command line switch **+vmm_log_default=DEBUG** will enable printing of all messages with the severity level **DEBUG** and higher. Other settings are **WARNING**, **NORMAL**, **TRACE**, or **VERBOSE**.

2.6.2 Using set_verbosity()

The method **vmm_log::set_verbosity()** allows you to set the level of printing on the fly. The following code sets the level to **DEBUG** for any vmm_log object with "Drv" in its name:

```
log.set_verbosity(log.DEBUG_SEV, "/Drv/", "/./", );
```

The regular expression **"/./"** matches any string, so the following matches all vmm_log objects, regardless of their name or instance name:

```
log.set_verbosity(log.DEBUG_SEV, "/./", "/./", );
```

Note that since **set_verbosity()** and **DEBUG_SEV** are part of the **vmm_log** class, they must be prefixed with a handle to that class.

Also note that this call overrides the **+vmm_log_default** switch, and only applies to current vmm_log objects, not any created afterwards.

2.7 Further Exploration

- You can create complex, multi-line messages using the vmm_log methods **start_msg()**, **text()**, and **end_msg()**.
- You can change the formatting of vmm_log by extending the **vmm_log_format** class and register an instance with the **vmm_log::set_format** method.

See the VMM for more information and examples.

3 Verification Environment

3.1 Overview

Your testbench goes through many phases of execution, from initialization, simulation, and cleanup. The class **vmm_env** helps you manage these steps and ensures that all execute in the proper order.

Coding Guideline:
The **new()** method should only initialize values, and should never have any side effects such as spawning threads or consuming time. If a testbench object starts running as soon as **new()** is called, you will not be able to delay its start, or synchronize it with other testbench operations.

3.2 The Nine Steps

The vmm_env class divides a simulation into the following nine steps, with corresponding methods:

- **gen_cfg()** – Randomize test configuration descriptor
- **build()** – Allocate and connect test environment components
- **reset_dut()** – Reset the DUT
- **cfg_dut()** – Download test configuration into the DUT
- **start()** – Start components
- **wait_for_end()** – End of test detection
- **stop()** – Stop data generators and wait for DUT to drain
- **cleanup()** – Check recorded statistics and sweep for lost data
- **report()** – Print final report

3.3 Simplest Example

Above all these methods is **run()** which keeps track of which steps have executed, and, when called, runs the remaining ones. For example, the following program runs all nine steps automatically:

```
program test;
  initial begin
    verif_env env;
    env = new(...);
    env.run();
  end
endprogram
```

The class **verif_env** extends **vmm_env**. You call **run()** and it will call all the steps which have not yet been run.

3.4 Basic Example

The next example runs the first step, makes a modification to the configuration, and then completes the test:

```
program test;
  initial begin
    verif_env env;
    env = new(...);

    env.gen_cfg();              // Create rand config
    env.rand_cfg.n_frames = 1; // Only run for 1 frame
    env.run();                 // Run the other steps
  end
endprogram
```

3.5 Automatic Sequencing

The **run()** task is not the only method that executes the steps. As shown in the following example, if you call **build()** without calling **gen_cfg()**, the **build()** method will automatically execute the previous step:

```
class my_eth_fr extends eth_frame;
  rand bit [47:0] mac_address;
  constraint one_port_only {
    da == mac_address;          // Use fixed address
  }
endclass

program test;
  initial begin
    verif_env env
    env = new();
    env.build();                  // Config and build
    begin
      my_eth_fr my_fr;
      my_fr = new();              // Use my own frame
      void = my_fr.randomize();
      env.src[0].rand_fr = my_fr; // Use to build more
    end
    env.run();
  end
endprogram
```

3.6 Using vmm_env

The following example defines the class **verif_env**. The virtual methods **gen_cfg()** and **build()** **must** call their super method as the first step. These calls to the base methods contain the sequencing code that ensures all previous steps have been called. If you leave out the calls, the **vmm_env** class will generate a fatal error at run time.

```
`include "vmm.sv"
```



```
class verif_env extends vmm_env;
  my_cfg cfg;
  my_gen gen[4];
  my_drv drv[4];
  my_mon mon[4];

  virtual function void gen_cfg();
    super.gen_cfg();
    // rest of gen_cfg method
  endfunction

  virtual function void build();
    super.build();
    // rest of build method
  endfunction
endclass
```

3.7 Detailed Explanation of Methods

3.7.1 gen_cfg()

This method creates a random configuration of the test environment and DUT. It may choose the number of input and output ports in the design and their speed, or the number of drivers on a bus and their type (master or slave). You can also randomly select the number of transactions, percent errors, and other parameters. The goal is that over many random runs, you will test every possible configuration, instead of the limited number chosen by directed test writers.

3.7.2 build()

This method builds the testbench configuration that you generated in the previous method: generators and checkers, drivers and monitors, and anything else not in the DUT.

3.7.3 reset_dut()

This method resets the DUT to make it ready for configuration.

3.7.4 cfg_dut()

In this method you download the configuration information into the DUT. This might be done by loading registers using bus transactions, or backdoor loading them using C code.

3.7.5 start()

This method starts the test components. This is usually done by starting the transactor objects, which will be described in section 8.

3.7.6 wait_for_end()

This method waits for the end of the test, usually done by waiting for a certain number of transactions or a maximum time limit.

The following example shows the **wait_for_end()** method with its inner wait for the end of test event, plus a time-out statement. These are wrapped in a **fork-join_any** that completes when either of these complete, and then a terminate command to stop the unfinished statement/threads.

```
class verif_env extends vmm_env {
   virtual task wait_for_end();
      super.wait_for_end();   // Call super method
      fork                      // Limit scope of terminate
      begin
         fork
            sync(ALL, this.sb.enough); // End of test event
            begin
               delay(TIME_OUT);
               `vmm_fatal(log, "Test did not complete");
            end
         join any
         terminate;
      end
      join
   endtask
```

3.7.7 stop()

This method stops the data generators and waits for the transactions in the DUT to drain out.

3.7.8 cleanup()

Check recorded statistics and sweep for lost data.

3.7.9 report()

Print the final report. Note that **vmm_log** will automatically print its report at the end of simulation so you do not have to write any special code for this.

3.8 Lab 1 vmm_env

Using the lab instructions (Appendix B), complete Lab 1 for vmm_env.

4 Data and Transactions

4.1 Introduction

Traditionally, when you created a testbench in Verilog, you implemented transactions as procedures, one per transaction. This caused the following problems:

- Code is not self-contained
- Code is not protected properly
- Cannot extend data types
- Cannot add constraints to an existing data type

Instead, you should model transactions as objects. Their data values exist in a transaction class that can be randomized, copied, packed, and unpacked. The code that actually executes the transactions resides in the Driver.

4.2 Transaction Coding Guidelines

Properties in a transaction class should be public so they can be modified or constrained by other classes, such as the testbench. Do not hide data values using set() & get() methods. In hardware verification, you need access to all parts of the testbench for maximum control.

Properties should be random by default so that they can be randomized by other classes. You can always go back and use **`rand_mode()`** to turn this off.

4.3 Transactions vs. Transactors

The transaction class contains both physical values that are sent to the DUT (address, data, etc.) and meta-data that has extra information about the transaction, such as a "kind" field. Even though this might be encoded in the physical values, put it into a separate field that can be easily accessed and can be randomized.

```
enum kind_t = READ, WRITE;
class apb_transaction extends vmm_data;
  rand kind_t kind;
  rand bit [ 7:0] sel;
  rand bit [31:0] addr;
  rand bit [31:0] data;
endclass
```

The Transactor contains the code to send the transaction to the next testbench level. The following is a Driver to read and write to a real bus:

```
class apb_master;
  task do(apb_transaction tr);
    case (tr.kind)
      READ:
        tr.data = this.read(tr.addr);
      WRITE:
        this.write(tr.addr, tr.data);
```

```
    endcase
  endtask
endclass
```

4.4 Creating Your Own Transactions

The VMM recommends that you extend the VMM classes to create your own company-specific classes to build a buffer between the VMM and end users. In this buffer, you can customize the classes for your own best practices. This layer is most commonly added around vmm_data, though this tutorial uses the class directly.

4.5 ID Fields

Every transaction has three integer ID fields uniquely identifying it. The **stream_id** tells which stream created this object – useful when there are multiple generators. The **sequence_id** is used when a stream generator creates groups of related transactions, and identifies the group. The **object_id** identifies individual transactions in a sequence. In the following example, a constraint block uses the **stream_id**.

```
class vmm_data;
  integer stream_id;
  integer scenario_id;
  integer object_id;
  ...
endclass

class atm_cell extends vmm_data;
  rand integer has_vlan;
  …
endclass

class my_atm_cell extends atm_cell;
   ...
   constraint stream_0_is_vlan {
      if (stream_id == 0) has_vlan == 1;
   }
endclass
```

4.6 Constraints

Every transaction should have one or more constraint blocks for the "must-obey" constraints that are never turned off or overridden. For example, they would make sure an integer field is never negative or that a length field is never 0. Name these constraints "*class_name*_valid", such as **atm_cell_valid**.

You should have separate constraints for "should-obey". You can turn these off later for injecting errors. Name these constraints "*class_name_rule*".

4.7 Methods

The vmm_data class defines a set of virtual methods for manipulating its properties. (See Appendix A for a review of virtual methods.) Here are some of the ones used in this tutorial. You will need to make your own methods when you extend **vmm_data**.

4.8 display() & psdisplay()

These methods display the contents of the class, either to the screen or to a string.

```
virtual task              display (string prefix);
virtual function string   psdisplay(string prefix);
```

4.9 allocate()

This method allocates a **vmm_data** object and initializes required fields. This is a virtual method, unlike **new()** so the correct method is called regardless of the handle type.

```
virtual function vmm_data allocate();
```

4.10 copy()

This method makes a copy of an existing transaction. It has an optional "to" field so you can copy to a previously allocated object. Note that this method returns a vmm_data type, so you may need to use **$cast()** with it.

```
virtual function vmm_data my_data::copy(vmm_data to = null);
   my_data cpy;

   // Copying to a new instance?
   if (to == null)
      cpy = new();
   else

   // Copy to an existing instance. Is it the correct type?
   // If destination handle is passed as argument during copy()
   // call, use $cast to check that handle type is correct.
   // Argument handle has to be base type (vmm_data) or my_data
   // else abort.  IF argument is base type(vmm_data), cpy
   // handle is upcasted to vmm_data type (cpy -> "to").

   if (!$cast(cpy, to, CHECK)) begin
      `vmm_fatal(this.log,
          "Attempting to copy to a non my_data instance");
      return;
   end

   // Copy ID's and any other properties
   cpy.stream_id   = this.stream_id;
   cpy.scenario_id = this.scenario_id;
   cpy.object_id   = this.object_id;


   // Assign the copy to the return handle
```

```
    copy = cpy;
endfunction
```

There is an alternative to the **copy()** method. The **new** keyword does a shallow copy – more of a "photocopy" of variable values. Encapsulated objects are not copied; the handle value is just copied. Note that the **new()** method is not called, so the result will have the same ID fields as the original.

The following example shows how to use the copy method:

```
my_data d1, d2, d3;  // Handles
d1 = new();          // Allocate object
d2 = d1;             // Both d1 & d2 point to same object
d2 = d1.copy();      // Error, copy returns vmm_data
$cast(d2, d1.copy()); // Good
void = d1.copy(d2);  // Copy d1 contents to d2 object
void = d1.copy(d3);  // Bug - d3 is still null
```

4.11 compare()

This method compares two objects and reports the difference.

```
virtual function bit compare (to, diff, kind);
```

The current object is compared with "to" using type "kind". (See 4.3 for an example of "kind".) The method returns 1 if the two objects are the same, 0 if not. The diff string gives a description of the difference.

4.12 Packing and unpacking

The following three methods are used for converting between the physical fields of an object and an array of bytes:

```
virtual function integer  byte_size  (kind);
virtual function integer  byte_pack  (bytes,offset,kind);
virtual function integer  byte_unpack(bytes,offset,kind);
```

The method **byte_size** tells how many bytes you need to pack an object of this kind. The method **byte_pack** packs the object of type kind into a dynamic array of bytes. The method **byte_unpack** unpacks the data from the dynamic array of bytes. The **offset** tells the methods where to start in the byte array.

4.13 Lab 2 – the vmm_data class

Using the VMM Basic lab instructions, complete Lab 2 for vmm_data.

5 Notification

5.1 Introduction

VMM provides an event notification class that allows you to notify when an event notification is indicated, and includes data. These notifiers are based on integer identifiers that hold a symbolic value. For example, the following code creates three notification identifiers associated with the **atm_driver** class:

```
class atm_driver extends vmm_xactor;
   static integer TR_STARTED;
   static integer TR_ABORTED;
   static integer TR_SUCCESS;
endclass
```

You must configure a notifier before using it. A notifier can be **ON_OFF**, **ONE_SHOT**, or **BLAST**. The following example calls the **configure()** method in the notify object (pre-instantiated in **vmm_xactor**, described in a later section), which returns an integer value.

```
class apb_master extends vmm_xactor;
  function new(...);
    this.TR_STARTED = this.notify.configure(*,
                                this.notify.ON_OFF_TRIGGER);
  endtask
endclass
```

When you indicate an event, you can optionally attach an object derived from **vmm_data**. You use this to describe why you indicated the notification. You should not modify this object. If you need to change the object, use callbacks instead, shown later in this tutorial.

```
forever begin
   atm_cell cell;
   ...
   this.pre_cell_tx_t(cell, drop);
   foreach (callbacks[i]) begin
      ...
   end
   if (drop) continue;
   this.notify.indicate(this.PRE_CELL_TX, cell);
   ...
end
```

5.2 Pre-defined Events

Many VMM classes include notification service interfaces, instances of **vmm_notify**. As shown above, **vmm_data** has an instance of **vmm_notify** and the events **EXECUTE**, **STARTED** and **ENDED**. The **vmm_xactor** class includes the notify properties **XACTOR_IDLE**, **XACTOR_BUSY**, **XACTOR_STARTED**, **XACTOR_STOPPED**, and **XACTOR_RESET**.

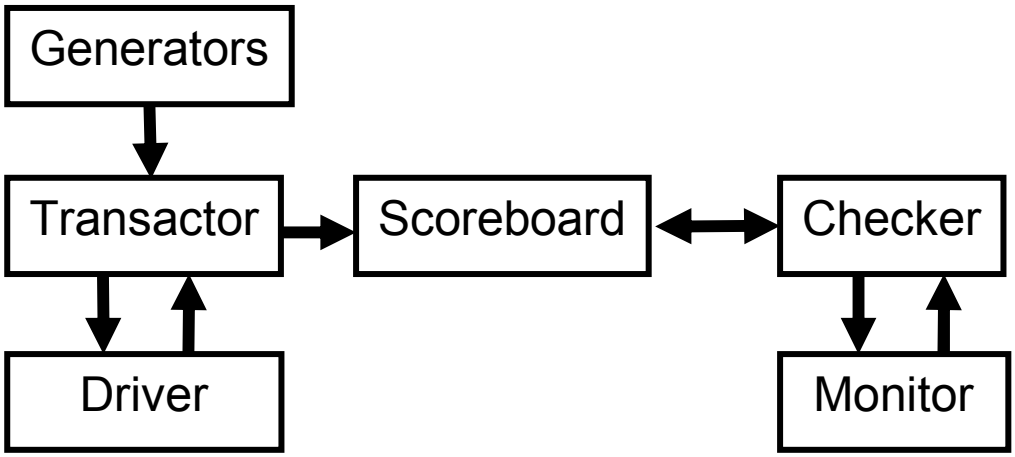
5.3 Further Exploration

When you use a VMM notification to pass an object, it should be treated as read-only. To pass an object that may be modified in more than one location in your testbench, use the **`vmm_broadcast`** class. This can buffer objects so you will not lose data.

6 Channels and Completion Models

6.1 Introduction

Your testbench environment needs to exchange transactions between its components. For example, transactions flow from the Generator -> Transactor -> Driver, or from the Monitor -> Checker -> Scoreboard. As you saw in the last section, you should model transactions as objects that are then created and modified by the different testbench components.



The connection between these components is the VMM channel. One side is the producer (such as the Generator) putting transactions into the channel. The consumer side (Transactor) gets the transactions out of the channel, and executes them.

The VMM channel has several advantages over the SystemVerilog mailbox:

- Unlike mailboxes, channels are strongly typed which helps prevent coding errors.
- Channels allow flow control, so the **`put()`** method will block if the channel is full.
- A channel can have both high-water and low water marks to fine tune the interactions between the producer and consumer. The **`get()`** method removes the transaction from the end of the channel, while **`peek()`** give you a handle to it without removal. Both block if the channel is empty.
- The output of a channel can be replicated using the **`tee()`** method.

6.2 Definition and Creation

You define a channel for a specific type using a macro, and then create channels as shown:

```
class atm_cell extends vmm_data;
   ...
endclass

// macro automatically creates new data type by
// appending "_channel" to data_type_name
vmm_channel(atm_cell)    // atm_cell_channel declaration
```

```
program test;
  initial begin
    atm_cell_channel chan;
    chan = new("ATM Cell channel", "class");
  end
endprogram
```

Now two threads can create communicate as follows:

```
// Producer
forever begin
  atm_cell cell = new();
  ch.put_t(cell);
end
```

and:

```
// Consumer
forever begin
  atm_cell cell = ch.get();
  ...
end
```

6.3 Under the Hood

Just like a SystemVerilog mailbox, the channel contains handles to objects, not the object themselves. You can modify an object after it has been put in the channel, leading to a common mistake:

```
// Producer
atm_cell_channel ch = new("ATM Cell channel", "class");
atm_cell cell;
cell = new();
while (...) begin
   void = cell.randomize();
   ch.put_t(cell);
end
```

This code only allocates a single cell. It then repeatedly randomizes this cell and puts it in the channel. The result is that the channel will contain many references to the same object. The solution is to allocate a new cell every time through the loop.

```
while (...) begin
   cell = new();
   void = cell.randomize();
   ch.put_t(cell);
end
```

6.4 Using Channels to Connect Blocks

The most common way to use channels is to allocate them in the **vmm_env** object and then pass them into the testbench components as shown in the following example.

```
task dut_env::build() {
   chan = new(...);
```



```
   consumer = new(chan);
   producer = new(chan);
}
```

6.5 Transaction Completion

You can synchronize two testbench blocks using a channel. The easiest way is to configure the channel with **full=1** (the default) so it works like a procedural interface. The producer thread blocks when it calls **put()**. When the consumer calls **get()** to remove the transaction from the channel, the producer unblocks so it can create a new transaction. In the following example, the consumer first calls **peek()** to read the transaction, but does not call **get()** until it is done, thus waking the producer:

```
// Consumer
forever begin
   cell = ch.peek();      // Read the cell
   case (cell.kind) {
      ...                 // Process the cell
   }
   void = ch.get();       // Done, wake up producer
end
```

6.6 Further Exploration

See the VMM section on "Completion and Response Models" for more examples of synchronizing using channels. What if you have more than one producer or consumer? The **vmm_broadcast** class is used for one-to-many communication, while **vmm_scheduler** takes N inputs and combines them into a single output.

7 Atomic Generators

7.1 Introduction

Tests should tune random generators, not completely rewrite them. This results in less code (each test is smaller), more randomness (all unspecified behavior is random) and more checking (extra randomness broadens the stimulus). Traditional testbenches create stimulus with generators that grow more and more complex as the project progresses, accommodating every variant of stimulus, error generation, synchronization, etc. This "mother of all generators" can be unstable because of the constant changes, as well as difficult to enhance and maintain. VMM recommends a different approach.

7.2 Adding Constraints

A typical generator might look like the following:

```
class cell_gen extends vmm_xactor;
   ...
   task main();
      forever begin
         atm_cell cell = new();
         void = cell.randomize();
         this.chan.put_t(tr);
      end
   endtask
endclass
```

The problem with this generator is that there is no easy way to randomize cells with different constraints.

CODING TIP: While the above example ignores the result from **`cell.randomize()`**, you should never do this in real code. Always check the result and issue a **`vmm_error`**. Otherwise, you may miss a constraint failure, leading you to think that the test generated cases that it really did not.

- You could use the **`randomize() with{}`** construct, but this would require a separate generator for each test, just what you were trying to avoid.
- You could modify the transaction class, **`atm_cell`**, adding constraints for each test, but this moves the problem to a different file.

Each of these requires every test writer to edit a common file, with the results applied to every generator / ATM cell. Some testbenches have knobs to control the different distributions and cases, but once again, the generator or transaction becomes the bottleneck, growing in complexity. In addition, the testcase is the testbench plus knob files, adding another file to the flow.

7.3 Factories

As an alternative, consider an automobile factory. It creates a stream of cars, each unique with varying options. The manufacturer accomplishes this by having a set of blueprints for the major variants (coupe, sedan, station wagon), and then allowing the user to tweak the details. You can use the same idea with creating transactions. Create a "factory" that stamps out transactions, then have individual tests feed it different blueprints. All the test-specific code is located in the test file, not the generators. In fact, you can have multiple factories running in a test, each generating a unique set of stimulus.

What does this look like? The following example uses a transaction class that extends vmm_data. The blueprint instance is randomized, then copied to a new instance processed by the next testbench layer.

```
class factory;
   transaction blueprint = new();
   task run();
      while (run) begim
         transaction tr;
         void = blueprint.randomize()
         $cast(tr, blueprint.copy());
         process(tr);
      end
   endtask
endclass
```

You can change the blueprint from the test level, which is just a SystemVerilog program:

```
class my_transaction extends transaction;
   constraint address_even {
      addr[0] == 0;
   }
endclass

program test;
  verif_env env;

  initial begin
    env = new();
    env.build();
    begin
       my_transaction my_tr = new();
       env.src[0].blueprint = my_tr;
    end

    env.run();
  end
endprogram
```

With this change, the generator **`src[0]`** will always create transactions with even addresses.

7.4 Benefits

Tests can modify constraints by

- Making variables non-random
- Turning constraint blocks off
- Add new constraint block
- Re-define constraint blocks
- Add random variables
- Supersede random results with directed data

Factory generators are:

- Good: Unmodified generic, reusable data class
- Good: Unmodified generic, reusable generator class
- Good: Different generators can have different constraints
- Good: Constraint set can be dynamically changed
- Good: Localized test-specific code
- Bad: Difficult to share constraint sets between testcases

7.5 Atomic Generator Macro

VCS includes a macro to create an atomic generator:

```
vmm_channel(atm_cell) //Macro: defines atm_cell_channel
vmm_atomic_gen(atm_cell, “ATM_Cell”) //Macro: defines atm_atomic_gen

class atm_env extends vmm_env;
  atm_cell_atomic_gen gen1; // Generator instance
  atm_cell_channel chan;    // Channel instance

  virtual function void build();
    chan = new(“Channel”, "from gen");
    gen1 = new(“Gen”, *, chan);      // Connect channel
    drv = new (“Driver”, “*”, chan); // to next level
  endfunction
endclass
```

The macro uses the blueprint object **`randomized_obj`**.

7.6 Lab 3 – Channels and the Atomic Generator

Using the VMM Basic lab instructions, complete Lab 3 for Channels and the Atomic Generator

8 Transactors

8.1 Introduction

By this point, you have seen VMM transactors. In this section, you are going to learn more about how to create and use them in simulation.

At its simplest, an VMM transactor is just a while loop that reads in transactions from a previous testbench layer, does some processing, and sends out transactions to the next layer. The key is properly starting and stopping the transactors.

The VMM has several transactor types:

- Active Xactor – Master
- Drives pins, blocks on channel get()
- Reactive Xactor – Slave
- Monitors and drives pins, blocks on signal edges
- Passive Xactor – Monitor
- Monitors pins, blocks on signal edges
- Also – Generator or other Xactors as needed
- Creates transactions, blocks using notification or channel put()

8.2 A Basic Transactor

Here is a basic transactor. You add code to method main() to process transactions. The other methods all start with a call to the base method to start and stop this method. For example, **`vmm_xactor::start_xactor()`** starts the virtual method main() – you don't need to do this.

```
class driver extends vmm_xactor;
   //start_xactor starts the execution threads
   virtual task start_xactor;
      super.start_xactor();
      ...
   endtask

   //stops execution threads after currently executing
   //transaction had completed. Takes effect at next call
   //to ::wait_if_stopped()
   virtual task stop_xactor();
      super.stop_xactor();
      ...
   endtask

   //resets the xactor's state and execution threads
   virtual task reset_xactor(...);
      super.reset_xactor(...);
      ...
   endtask
```

```
   virtual task main();
      ...
   endtask
endclass
```

8.3 Stopping and Starting

The **main()** method periodically checks to see if the transactor has been stopped by calling **wait_if_stopped()** as shown below:

```
virtual task main();
   forever begin
      this.wait_if_stopped();
      atm_cell cell = to_driver.get();
      this.wait_if_stopped();
      ...
   end
endtask
```

The **wait_if_stopped()** method will block if **stop_xactor()** has been called. Different blocks in your testbench will define when to stop and what it means. You should check if the transactor needs to stop after every time-consuming action, such as the call to **get()** above.

8.4 Physical and Virtual Interfaces

The SystemVerilog interface groups all relevant physical signals (ports) together, just as a C typedef combines several objects into a struct. A virtual interface is just a pointer to a physical interface. You can pass a virtual interface into drivers and monitors. Now the testbench can replicate a driver, with each instance using a separate virtual interface so as to drive multiple physical ports.
If you need to synchronize on a clock edge in an interface, use the clocking block in the interface.

```
@1 bus_ifc.cb;
```

Note that this form does not contain the active edge of the clock signal. If the designer changes the edge, you only have to change the interface definition, not every usage of the signal.

8.5 Reusable Transactors

Recall that one of the goals of VMM is that the testbench objects should not have to change. Test specific code goes in the test, not in the transactors such as the driver. The question then, is, how do you write a transactor that can meet all verification requirements such as injecting errors and delays, sampling data for functional coverage and connecting the scoreboard?

Before the VMM, the testbench would have the “mother of all transactors” (MOT) that performed all these actions and more. However, any time one would dream up a new way to control the DUT, one would have to edit the MOT. All these edits make the MOT unstable and a bottleneck for the verification team.

Instead, a transactor should do the simplest things by default – no errors, no delays, but has hooks so that you can add test-specific extensions. You can accomplish this by using callback methods. These are extensible virtual methods that are empty by default.

```
task main();
  ...
  forever begin
    ...
    cb.pre_drive_callback();

    // Drive out the transaction

    cb.post_drive_callback();
  end
endtask
```

In the above code, the method **cb.pre_drive_callback()** is called just before that transaction is driven into the design, and **cb.post_drive_callback()** is called just after the transaction is driven. By customizing your own method, you could insert delays, modify or even drop the transaction, gather functional coverage information on the transmitted transaction, and put this transaction into the scoreboard.

8.6 Creating Callbacks

Once you have identified key points in the transactor flow to insert callbacks, you can define a callback façade class:

```
class atm_callbacks extends vmm_xactor_callbacks;
   virtual task pre_cell_tx_t(atm_driver xactor,
                              atm_cell   cell,
                              var bit    drop)
    …
  endtask
endclass
```

The driver now just calls **pre_cell_tx()** before driving the cell. Next, define your test-specific callback classes such as **error_inject**, **scoreboard_insert**, **functional_cov** that extend **atm_callbacks**. Each callback class is then registered at the test level. The callbacks are called in the order you registered them. So, be sure to inject errors first, and then add the transaction to the scoreboard.

```
class stretch_ifg extends atm_callbacks;
   // This class stretches the inter-frame gap
   ...
endclass

program test;
  verif_env env;
  initial begin
    env = new();
    env.build();
```

```
    begin
      stretch_ifg cb = new();
      env.driver[0].prepend_callback(cb);
      env.driver[3].prepend_callback(cb);
    end
    env.run();
  end
endprogram
```

You can see that the code to invoke the callbacks is common across all transactors, so the VMM provides a macro:

```
class atm_driver extends vmm_xactor;
   ...
   forever begin
      atm_cell cells = this.in_chan.get();
      bit drop = 0;

      `vmm_callback(atm_driver_callbacks,
                    pre_cell_tx_t(this, cell, drop));
      if (drop) continue;

      // Drive the cell here …

      `vmm_callback(atm_driver_callbacks,
                    pre_cell_tx_t(this, cell, drop));
   end
endclass
```

8.7 Labs 4, 5,6and 7

Using the VMM Basic lab instructions, complete Lab 4 for the APB Master Transactor, Lab 5 for the Monitor Transactor, and Lab 6, Scoreboard Integration

Lab 7, Functional Coverage, is optional. It shows an example of adjusting random constraints to increate functional coverage.

Appendix A OOP & Virtual Methods

8.8 Introduction to Classes

With traditional procedural programming, you create structs to hold the data, and global procedures that manipulate the data. With Object Oriented Programming, you wrap the data and the procedures inside of a class:

```
class packet;
   logic [31:0] src, dst, crc, data[8];

   task display();
      ...
   endtask

   function logic [31:0] compute_crc();
      ...
   endfunction
endclass

program test;
  packet p;
  initial begin
    p = new();
    p.display();
    p.crc = p.compute_crc();
  end
endprogram
```

8.9 Inheritance

Now that you have a packet, how do you create different flavors? For example, you might want to add an is_good bit that tells if the class should always generate good CRC. In OOP, this is done by extending the original packet class (base class):

```
class my_packet extends packet;
   bit is_good;
   function compute_crc();
      compute_crc = super.compute_crc();
      if (!is_good) compute_crc = urandom;
   endfunction
endclass
```

8.10 Handles to Objects

When you make the following declaration:

```
packet p;
```

you are creating a handle that can reference objects of type packet. The handle is initialized to null. You can call the new() method to create an object:

```
p = new();
```

This allocates enough memory to hold the class packet, 11 longwords (**sa**, **da**, **crc**, and **data[8]**). If you allocate a **my_packet** object, 12 longwords will be allocated (the original 11 plus one more for the **is_good** bit).

What happens if you try to mix handles for base and extended classes? The easiest way to visualize this is to consider the **is_good** bit. It would be an error if you used a **my_packet** handle to access the **is_good** bit, but the handle points to only a packet object (11 longwords).

```
packet p;
my_packet mp;

mp = new();
p = mp;     // Good: Copies a handle

p = new();
mp = p;     // Error: mp.is_good won't exist
```

But what if you use a base handle to point to an extended object, then try to assign back to an extended handle? This is normally not allowed, so you will need to use **$cast()**. This method checks the object type to make sure it matches the destination object.

```
packet p;
my_packet mp, m2;

mp = new();
p = mp;       // Still good
$cast(m2, p); // Good: p points to my_packet object
```

8.11 Polymorphism

What happens if you call a method in a class. By default, if you use a packet handle to call the **compute_crc()** method, the **packet::compute_crc()** method is called, even if the object was actually of type **my_packet**. (See code in the Inheritance section above.) But if you use virtual methods, VCS will call the method based on the object type, not the handle type:

```
class packet;
   virtual function logic [31:0] compute_crc();
      ...
   endfunction
endclass

class my_packet extends packet;
   bit is_good;
   virtual function compute_crc();
      compute_crc = super.compute_crc();
      if (!is_good) compute_crc = ~compute_crc;
   endfunction
endclass
```



```
packet p;
my_packet mp;

initial begin
  mp = new();
  p = mp;
  p.crc = p.compute_crc();  // Calls my_packet method
end
```

Why is this useful? You can write a generic class such as **`packet`** and use it in a testbench:

```
class protocol;
   ...
   task transmit(packet pkt);
      ...
      pkt.crc = pkt.compute_crc();
      off = vera_pack(bytes, off, pkt);
      // Transmit packet data
   endtask
endclass
```

If you call the **`transmit()`** method with a **`packet`** object, it will call **`packet::compute_crc().`** Call it with a **`my_packet`** object and it will call **`my_packet::compute_crc()`** and inject errors.

Appendix B: Labs

LAB 1: VMM Environment

Goal	Create a testbench environment class Issue a message
Location	lab1
Base Classes	vmm_env, vmm_log
Allocated Time	15 minutes

apb/apb_cfg.sv	APB Configuration Class
env/dut_env.sv	DUT Environment Class
tests/test_01.sv	Simple Test to run the environment

The DUT Environment class is the main top-level testbench class. This class instantiates all the permanent testbench elements, and controls the sequence of testbench steps. The environment is DUT specific, so it is in the env/ directory.

1) DUT Environment Class Code Review

   Review the dut_env.sv file and answer the following.

   How many steps are present in the vmm_env flow?_____
   Are the tasks/functions all defined as virtual?_____
   Why?_____

2) APB Configuration Class Code Review

   The apb_cfg class contains the random configuration details for the testbench. For simplicity, this consists of a single data item, to determine how many cycles to run before exiting the testbench.

   Review the code for the testbench configuration object in apb_cfg.sv.
   How many random integers are present in the class?_____

3) Adding APB Configuration Class to the APB Environment

   Edit the dut_env.sv file, and follow the directions in the file to accomplish the above steps. Each step has a "Lab1 – comment" to provide help in completing the task. The following steps add the config object to the APB environment:

a. Add a vmm_log handle to the dut_env class
b. Add a configuration handle to the dut_env class
c. Construct the log handle using new("dut", "env")
d. Construct the configuration handle
e. Randomize the configuration in the dut_env::gen_cfg() task
f. Add a debug print statement to the end of the dut_env::gen_cfg() task to print the value of the cfg.trans_cnt data using the `vmm_note() macro. Hint: Use $psprintf or $sformat

4) Compile and Run the Testbench

Compile the testbench using the following command in the lab1 directory.

```
csh% gmake lab1
```

Note the VCS command and options that are used by the Makefile here.
vcs

```
csh% gmake lab1
```

Run the testbench several times with random seeds.
Verify the trans_cnt value changes.

```
csh% ./simv +ntb_random_seed=1
csh% ./simv +ntb_random_seed=2
csh% ./simv +ntb_random_seed=3
```

Solutions for the above questions are at the end of this document.

LAB 2: Creating a vmm_data class

Goal	Create an APB transaction class
Location	lab2
Base Classes	vmm_data
Allocated Time	15 minutes

apb/apb_trans.sv	APB Transaction Class
Tests/test_02.sv	APB Transaction Test Program

The APB Data class will contain properties for an APB transaction. The properties include addr, data and an enumerated type for direction. This code is part of the reusable VIP, so it is in the apb/ directory. Each step has comments in the code to assist in editing.

1) Properties

The transaction class contains a random property for each data field.

For the APB protocol, add the following properties.

```
typedef enum {READ, WRITE} dir_e;
rand dir_e dir;
rand logic [31:0] addr, data;
```

2) copy()

The copy() function is used by a transaction to make a copy of itself.

For the APB protocol, review the copy() function contents.

3) copy_data()

The copy_data() function is used by a transaction to copy each data field.

Complete the copy_data() function by adding a line for each property.

4) compare()

The compare() function is used by a transaction to compare itself with another transaction. The pass/fail result is returned by the function, and a 'diff' string is used to return a compare message.

For the APB protocol, complete the compare() function by adding a block to compare the property to the end of the compare() function.

5) psdisplay()

The psdisplay() function is used to create a text string from a transaction for printing to the screen or log file.

Review the psdisplay() function and note the function uses several $sformat() statements to create a string. The string contains the transaction in a text format, including the stream_id, scenario_id and data_id values.

6) byte_size()

This function calculates the number of bytes in a transaction.

Review the code for byte_size(). In our simplified APB protocol, the size is always 4 bytes, so this function returns a constant.

7) byte_pack()

This function packs the transaction object into a byte-array.

Review the code for byte_pack(). In our simplified APB protocol, the transaction is always 4 bytes long, so the transaction can simply be packed with 4 assignment statements.

8) byte_unpack()

This function unpacks a byte-array of data into a transaction object.

Review the code for byte_unpack(). In our simplified APB protocol, the transaction is always 4 bytes long, so the transaction can be unpacked with an assignment statement.

9) Testing the apb_trans class

A short test is supplied to test the apb_trans class. Briefly review the teststs/test02.sv and note that the test performs the following tasks.

a. creates a random apb_trans
b. prints this transaction to the screen
c. copies the apb_trans
d. prints the copied transaction to the screen
e. compares the two transactions

Compile and run the test using the following command in the lab2 directory.

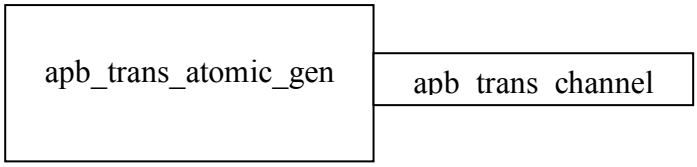
```
csh% gmake lab2
```

LAB 3: Channels and Atomic Generator

Goal	Create an atomic generator and add it to the environment
Location	lab3
Base Classes	vmm_data, vmm_atomic_gen macro
Allocated Time	15 minutes

apb/apb_trans.sv	APB Transaction Class
apb/apb_cfg.sv	APB Configuration Class
env/dut_env.sv	DUT Environment Class
tests/test_03.sv	Simple Test Program

In this lab, an atomic generator and channel will be added to the environment. The atomic generator will create streams of individually-randomized transactions. The apb-channel will be used to connect the generator to other transactors in the environment.



The APB Transaction class contains the following two macro statements.

```
`vmm_channel(apb_trans)
`vmm_atomic_gen(apb_trans)
```

These macro statements cause the following classes to be created automatically when the apb_trans.sv file is compiled.

```
class apb_trans_channel;
class apb_trans_atomic_gen;
```


1) Addition of the Atomic Generator and Channel

The macro statement creates the atm_trans_atomic_gen class when the apb_trans.sv file is compiled. This class can easily be used by the testbench to create a stream of individually-randomized transactions.

Edit the dut_env.sv file, and follow the directions in the file to accomplish the above steps. Each step has a "Lab3 – comment" to provide help in completing the task. The following steps are required to add an atomic generator to the environment.

a. Define an apb_trans_channel handle called gen2mas
b. Define an apb_trans_atomic_gen handle called gen
c. Create the channel by calling new() in the dut_env::build() task
d. Create the generator by calling new() in the dut_env::build() task
e. Configure the generator by adding gen.stop_after_n_inst = cfg.trans_cnt in the dut_env::build() task
f. Start the generator by calling gen.start_xactor() in dut_env::start()
g. Wait for the generator to signal completion (DONE) by calling gen.notify.wait_for() in the dut_env::wait_for_end() task
(hint: apb_trans_atomic_gen::DONE, Note: enum {DONE} is created when vmm_atomic_gen macro is called, please refer to VMM manual for more detail)
h. Stop the generator by calling gen.stop_xactor() in dut_env::stop()

2) Compile and Run the Testbench

Compile and run the testbench using the following command.

```
csh% gmake lab3
```

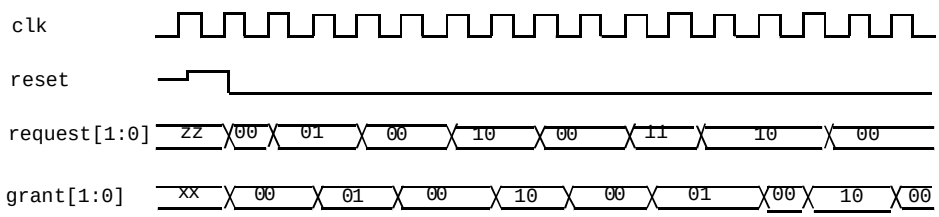
Note: All atomic generators contain the following properties.
- randomized_obj (blueprint)
- stop_after_n_insts(control)
- DONE (notify)
- GENERATED (notify)

LAB 4: APB Master Transactor

Goal	Create an APB Master Transactor
Location	Lab4
Base Classes	vmm_xactor, vmm_xactor_callbacks
Allocated Time	30 minutes

apb/apb_if.sv	APB Interface Definition
apb/apb_master.sv	APB Master Transactor
apb/apb_trans.sv	APB Transaction Class
env/dut_env.sv	Testbench Environment
tests/test_04.sv	Simple Test Program

A timing diagram for the APB interface is shown below for reference during this lab.



The APB master transactor extracts apb_trans objects from a channel, and executes read/write cycles on the APB bus. The goal of this lab is to create a VMM-compliant transactor, not to focus on the low-level physical protocol. For this reason, do_read(), do_write() and do_idle() tasks are provided to perform the bus cycles.

1) Create an APB Master Transactor

   Complete the new() task of the apb_master class, using the following arguments.

```
string instance;
integer stream_id = -1;
virtual apb_if.Master apb_master_mp;
apb_trans_channel in_chan = null;
```

2) Review the new() task

   Review the code in the new() task. Note that if the in_chan argument is not specified (null), a channel will be created automatically.

3) Review the main() task

Note the following:

a. super.main is called to perform any base-class actions
b. The body of main() is an infinite forever loop
c. A transaction is extracted from the channel
d. A pre-tx callback is called
e. The transaction is processed (read, write or idle cycles executed on the bus)
f. A post-tx callback is called

4) Add a post-tx callback

Add a `vmm_callback() macro call to the main() task after the transaction is processed. The arguments to the macro are detailed in the comments.

5) Review the reset(), do_read(), do_write(),do_idle() tasks

These tasks execute the various transactions on the physical bus. Note that these tasks contain code that is typically present inside a BFM model. These tasks block as needed, and are protocol specific.

6) Integration of APB Master into the APB Environment

The following steps are required to add the apb_master into the environment in dut_env.sv.

a. Define an apb_master handle called mst
b. Create the master by calling new() in the dut_env::build() task
c. Reset the master by calling mst.reset() in dut_env::reset_dut() task
d. Start the master by calling mst.start_xactor() in dut_env::start()
e. Stop the master by calling mst.stop_xactor() in dut_env::stop()

Edit the dut_env.sv file, and follow the directions in the file to accomplish the above steps. Each step has a "// Lab4" comment to provide guidance.

7) Compile and Run the Testbench

Compile and run the testbench using the following command in the lab4 directory.

```
csh% gmake lab4
```

Optional: View the waveforms using DVE.

```
csh% dve &
File > Open Database > vcdplus.vpd
```

LAB 5: APB Monitor Transactor

Goal	Create an APB Monitor Transactor
Location	Lab5
Base Classes	vmm_xactor, vmm_xactor_callbacks
Allocated Time	15 minutes

apb/apb_if.sv	APB Interface Definition
apb/apb_monitor.sv	APB Monitor Transactor
apb/apb_trans.sv	APB Transaction Class
apb/dut_env.sv	Testbench Environment
tests/test_05.sv	Simple Test Program

1) Review the APB Monitor Code

   Note that the Monitor is similar to a generator, as both create a stream of transactions. The monitor uses the factory pattern, to copy a blueprint in a similar fashion to the generator. This allows the test writer to replace the default transaction object with a custom object, containing additional tracking information or other properties.

   The monitor class contains a randomized_obj declaration that is constructed in the apb_monitor::new() task.

   The main() task contains an infinite loop, to constantly monitor the bus.
   The bus is monitored in the sample_bus() task containing BFM-like code.

2) Add the Factory/Prototype Pattern code

   In the main() task, add code to copy the randomized_obj instance, and cast assign this to the local 'tr' data variable. Comments in the code provide hints on how to accomplish this.

3) Add a Post-Rx Callback

   After the sample_bus() task is called, add a Post-Rx callback macro call to invoke the VMM callback class tasks. Comments in the code help with the required syntax; this will be very similar to the Post-Rx callback in the APB Master.

   Integration of APB Monitor into the APB Environment

The following steps are required to add the apb_monitor into the environment. Edit the dut_env.sv file, and follow the directions in the file to accomplish the above steps. Each step has a "Lab5 – comment" to provide help in completing the task.

a. Define an apb_monitor handle called mon
b. Define a dut_scb handle called scb
c. Create the scoreboard by calling new() in the dut_env::build() task
d. Create the monitor by calling new() in the dut_env::build() task
e. Start the monitor by calling mon.start_xactor() in dut_env::start()
f. Stop the monitor by calling mon.stop_xactor() in dut_env::stop()

4) Compile and Run the Testbench

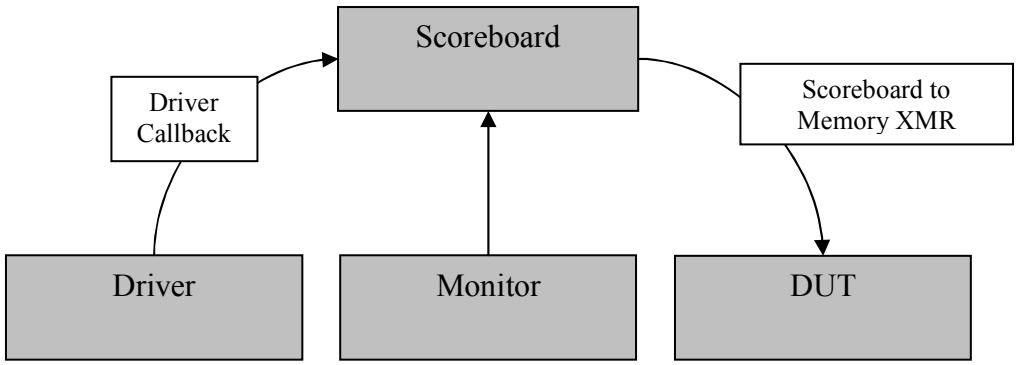
Compile and run the testbench using the following command in the lab5 directory.

```
csh% gmake lab5
```

LAB 6: Scoreboard Integration

Goal	Integrate the scoreboard using callbacks
Location	Lab6
Base Classes	vmm_xactor, vmm_xactor_callbacks
Allocated Time	15 minutes

`env/apb_scb.sv`	Scoreboard Class
`env/scb_callbacks.sv`	Scoreboard Callback Class
`env/dut_env.sv`	Test Bench Environment
`tests/test_06.sv`	Simple Test Program



The scoreboard is integrated using callbacks attached to the Driver and Monitor transactors. This allows the scoreboard to integrate with the APB VIP components in a passive manner, and requires no code changes or restrictions on the APB classes.

1) Review the APB Callback Class

   Review the apb/apb_master.sv file and verify following tasks are present.

   apb_master_callbacks::master_pre_tx()
   apb_master_callbacks::master_post_tx()

   As these classes are base-classes for the APB callbacks, the tasks are empty. The code to perform callback actions is DUT or test specific, so it does not appear in the APB VIP directory files.

2) Review the Scoreboard Callback Class and Integrate the Scoreboard

   Review the env/scb_callbacks.sv file.

Verify the following class::tasks are present

apb_master_callbacks::master_pre_tx()
apb_master_callbacks::master_post_tx()
apb_monitor_callbacks::monitor_pre_tx()
apb_monitor_callbacks::monitor_post_tx()

Add the following code

a. In master_post_tx, call scb.from_master() to add a transaction to the scoreboard
b. In monitor_post_tx, call scb.compare() to compare the received transaction to the expected one

3) Review the Scoreboard Class

Briefly review the env/dut_scb.sv file.
Note the following task to add data to the scoreboard from the master callback dut_scb::from_master(), and another task to compare the actual with expected dut_scb::compare().

Integration of Scoreboard into the APB Environment

The following steps are required to add the scoreboard into the environment. Edit the dut_env.sv file, and follow the directions in the file to accomplish the above steps. Each step has a "Lab6 – comment" to provide help in completing the task.

a. Create and construct an apb_master_scb_callbacks object called apb_mst_scb_cb, and append the callback to the APB Master object mst
b. Create and construct an apb_monitor_scb_callbacks object called apb_mon_scb_cb, and append the callback to the APB Monitor object mst
c. In wait_for_end, Wait for either the generator or scoreboard in a fork-join_any. The generator's wait is already there, just call scb.notify.wait_for(scb.DONE)
d. Clean up after the test by calling scb.cleanup() in dut_env::cleanup()
e. Create a report by calling scb.report () in dut_env::report ()

4) Compile and Run the Testbench

Compile and run the testbench using the following command in the lab6 directory.

```
csh% gmake lab6
```

LAB 7: Functional Coverage

Goal	Add functional coverage using callbacks
Location	Lab7
Base Classes	vmm_xactor, vmm_xactor_callbacks
Allocated Time	15 minutes

env/cov_callback.sv	Coverage Callbacks & Coverage Code
env/dut_env.sv	Test Bench Environment
tests/test_07.sv	Simple Test Program

The coverage will be integrated using callbacks attached to the Master transactor. This allows coverage to be added in a passive manner, with no changes to the APB master.

1) Add Coverage Points

The structure of the coverage callback class is similar to the scoreboard callback class. Significantly more code is present in the coverage callback, as the coverage model has been placed directly in the class.

Review the env/cov_callbacks.sv file, and add the following coverage points:

```
data: coverpoint tr.data {
  bins zero = {0};
  bins onek = {1024};
  bins others = default;
}
```

2) Integration of Scoreboard into the APB Environment

The following steps are required to add the scoreboard into the environment:

a. Define an apb_master_cov_callback cov_cb and call new()
b. Append the callback to the mst APB Master object

Edit the dut_env.sv file, and follow the directions in the file to accomplish the above steps. Each step has a "Lab7 – comment" to provide help in completing the task. This code will look very similar to the master scoreboard callback.

3) Compile and Run the Testbench

Compile and run the test using the following command in the lab7 directory

```
csh% gmake lab7
```

4) Create and View the Functional Coverage Reports

The unified report generator is used to create the reports. Run the report using the following command:

```
csh% gmake cov
```

View the report by opening the HTML files in the urgReport/ directory with a browser.

Or use the following commands for older-style HTML or text coverage reports:

```
csh% vcs –cov_report .
csh% vcs –cov__text_report .
```

5) Constrain the atomic generator to increase coverage

It is unlikely that totally random APB transactions will hit the specified coverage points unless we run a huge number of simulation cycles. Adding constraints into the test to increase the chances that interesting stimulus will be generated will help.

Modify the tests/test_07.sv file as follows:

a) Add a new class derived from apb_trans called my_apb_trans
b) Add a constraint into the new class to constrain addr and data (see comments)
c) In the environment initial block, create a new scope with begin … end
d) Create a new my_apb_trans object
e) Place this object into the atomic generator object gen

Re-run the test, and examine the log output and coverage reports.
The addr and data fields should now be constrained, and the coverage numbers should be higher than before.

```
csh% gmake cov
csh% vcs –cov_report .
```

Question/ Answers

Lab1:

How many steps are present in the vmm_env flow? 9
Are the tasks all defined as virtual? Yes
Why? So you can override the functionality in your own derived env class.

How many random integers are present in the class? 1

