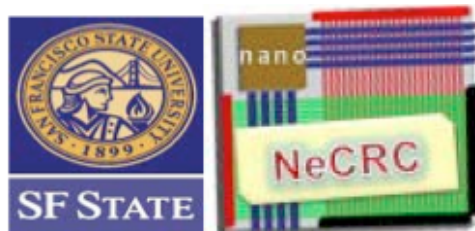


# **System Verilog Verification Methodology Manual (VMM 1.2)**

**Developed By  
Abhishek Shetty**

**Guided By  
Dr. Hamid Mahmoodi**

**Nano-Electronics & Computing Research Center  
School of Engineering  
San Francisco State University  
San Francisco, CA  
Spring 2012**



## Table of Contents

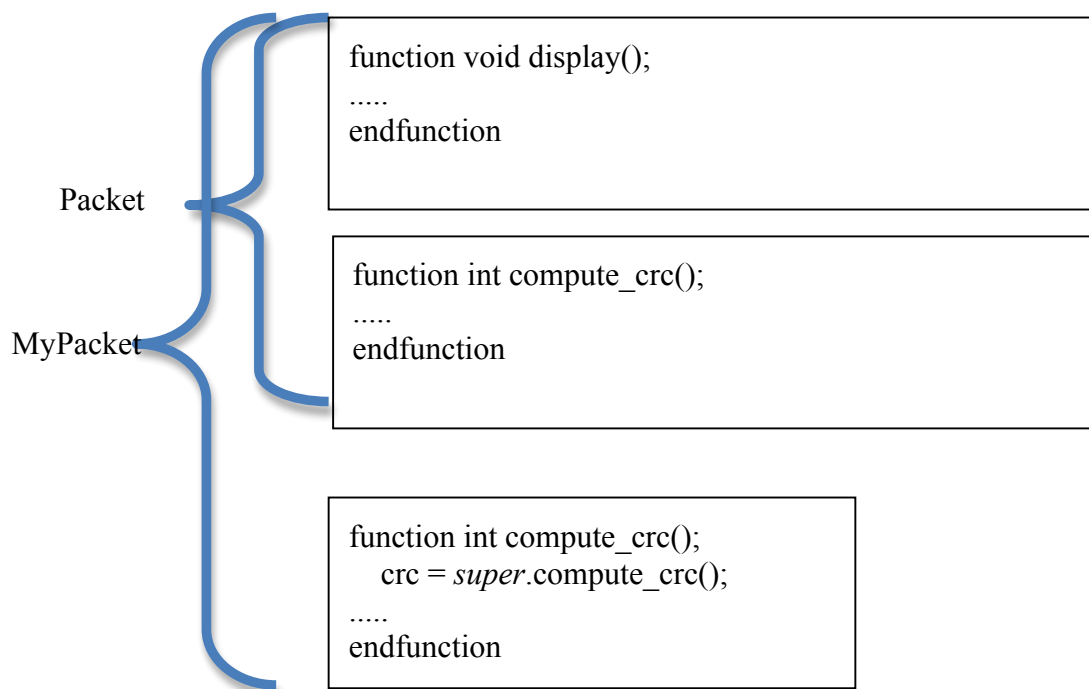
1. Object Oriented Language Basics .....	3
2. Layered Verification Environment .....	4
3. VMM Guiding Principles .....	6
4. Transactor Phasing .....	7
4.1 Explicit Phasing .....	9
4.2 Implicit Phasing .....	10
5. Transactor Callbacks .....	13
6. Scenario Generators .....	14
7. Creating Testbench Essential Elements .....	16
7.1 Transactor Class .....	17
7.2 Scoreboard Class .....	24
7.3 Coverage Class .....	18
7.4 Message Service .....	26
7.5 Creating Tests .....	27
8. Hands on creating of VMM testbench elements .....	28
9. Example DUT .....	30
10. Accessing examples from the Hafez server .....	36
11. References & Books .....	36

# 1. Object Oriented Language Basics

## Inheritance

- New classes derived from original base class
- Inherits all contents of base class
- Keyword *extends* used to inherit base class features

class My\_Packet *extends* Packet



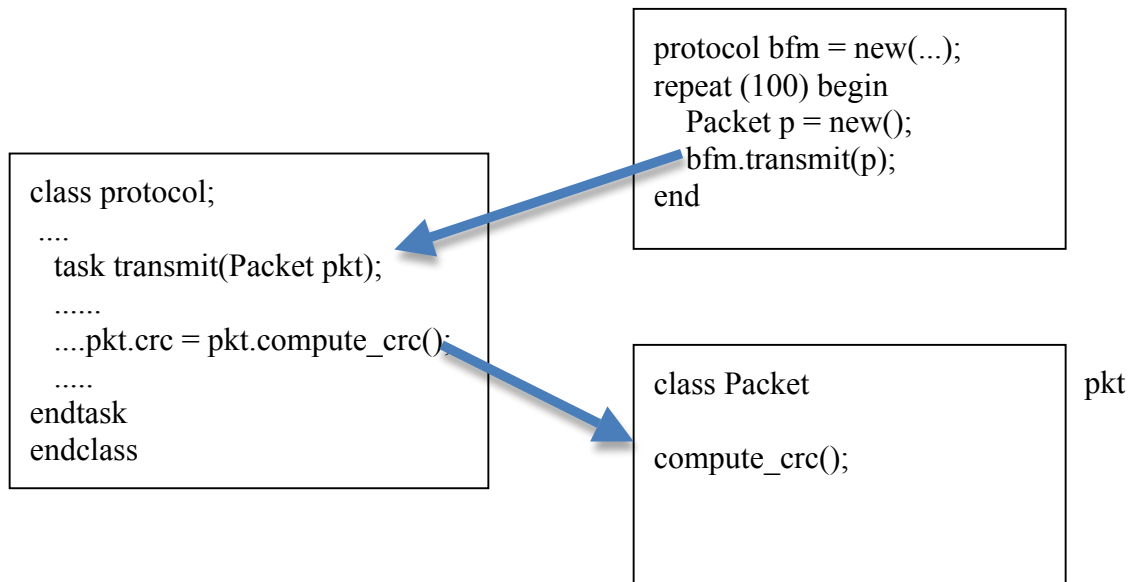
To over-ride the functionality of the compute\_crc in MyPacket extended class, keyword *super* is used to override the behavior from base class.

```
Packet P1;  
My_Packet P2;  
P1.compute_crc ();  
P2.compute_crc ();
```

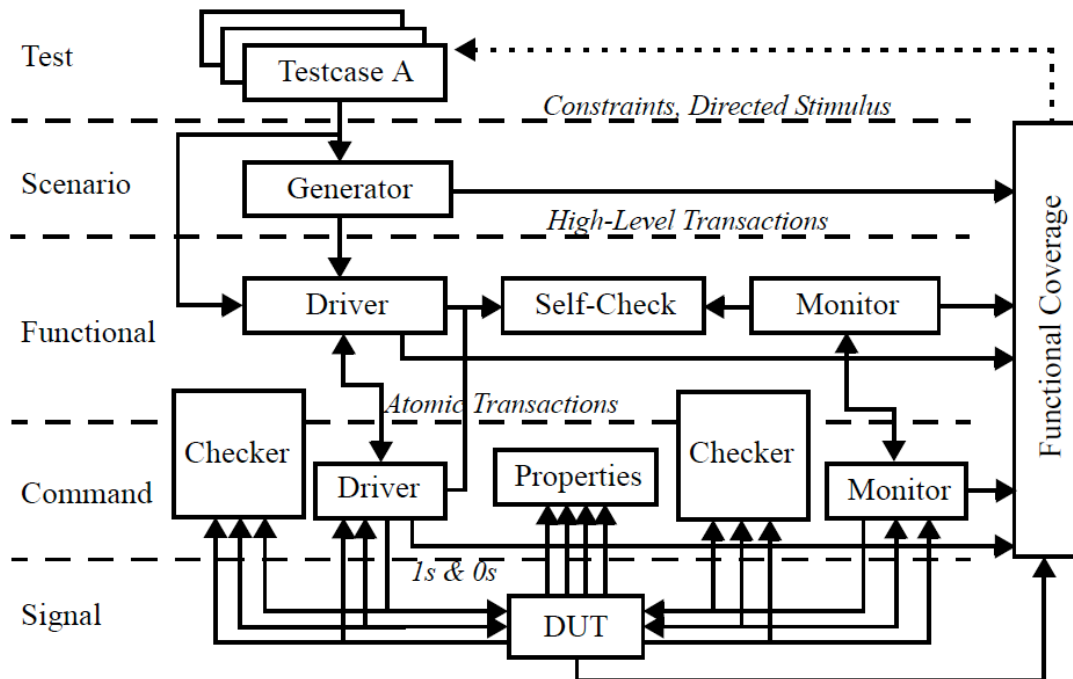
When a method within the scope of the object is declared to be *virtual*, then the last definition of the method in the object's memory will be executed.

## Polymorphism

The objects of the class inherit the properties and transfer it over to the other objects.



## 2. Layered Verification Environment



The DUT initiates the transaction and the reactive driver supplies the required data to successfully complete the transaction. For example, a program memory interface bus-functional model is a reactive driver. The DUT initiates read cycles to fetch the next instruction and the bus-functional model supplies new data in the form of an encoded instruction.

The term Transactor is used to identify components of the verification environment that interface between two levels of abstraction for a particular protocol or to generate protocol transactions. In above figure, the boxes labeled Driver, Monitor, Checker and Generator are all transactors. The lifetime of transactors is static to the verification environment. They are structural components of the verification components and they are similar to the DUT modules. A driver actively supplies stimulus data to the DUT. A proactive driver is in control of the initiation and type of the transaction.

A monitor reports observed high-level transaction timing and data information. A reactive monitor includes elements to generate the low-level handshaking signals to terminate an interface and successfully complete a transaction.

Both transactors and data are implemented using the *class* construct. The difference between a Transactor *class* and a data *class* is their lifetime. Limited number of Transactor instances is created at the beginning of the simulation and they remain in existence throughout. This creates a very large number of data and transaction descriptors instances throughout the simulation and they have a short life span.

Eg: Transactor Declarations

```
class mii_mac_layer extends vmm_xactor;  
...  
endclass: mii_mac
```

The **vmm\_xactor** base class contains standard properties and methods to configure and control transactors. To ensure that all transactors have a consistent usage model, you must derive them from a common base class.

### 3. VMM Guiding Principles

- Top-down implementation methodology
  - Emphasizes “Coverage Driven Verification”
- Maximize design quality
  - More testcases
  - More checks
  - Less code
- Approaches
  - a. Reuse
    - Across tests
    - Across blocks
    - Across systems
    - Across projects
  - b. One verification environment, many tests
  - c. Minimize test-specific code

#### VMM Base Classes and Macros

##### Base Classes

vmm\_test  
vmm\_group  
vmm\_data  
vmm\_xactor  
vmm\_tlm  
vmm\_env  
vmm\_subenv

##### Class Libraries

vmm\_log  
vmm\_opts  
vmm\_notify  
vmm\_broadcast/scheduler  
vmm\_ms\_scenario\_gen/vmm\_ms\_scenario

##### Class Macros

vmm\_channel  
vmm\_atomic\_gen  
vmm\_scenario\_gen

## Code Macros

vmm\_member\_begin/end  
vmm\_callback  
vmm\_fatal  
vmm\_error  
vmm\_warning  
vmm\_note  
vmm\_trace  
vmm\_debug  
vmm\_verbose  
vmm\_unit\_config  
vmm\_rtl\_config

The above listings are the collection of classes and macros most commonly used. The full listing of all available classes and macros are found in VMM Standard Library User Guide.

## 4. Transactor Phasing

Transactors progress through a series of phases throughout simulation. All transactors are synchronized so that they execute their phases synchronously with other transactors during simulation execution.

VMM supports two levels of Transactor phasing, implicit and explicit phasing. During explicit phasing, all the transactors are controlled by the master controller such as vmm\_env to call the Transactor phases. In implicit phasing, the transactors execute their phases automatically and synchronously.

In an implicitly phased testbench, functions and tasks representing phases are called automatically at the appropriate times. A global controller (vmm\_simulation) works in conjunction with specially configured schedulers (vmm\_timeline) to walk all testbench components through relevant phases. vmm\_simulation acts like a conductor, keeping all of the various testbench components in sync during pre-test, test, and post-test portions of a typical simulation.

Explicit Phase	Implicit Phase	Intended Purpose
gen_cfg	rtl_config gen_config	Determine configuration of the testbench
build	build	Create the testbench
	configure	Configure options
	connect	Connect TLM interfaces, channels
	configure_test_ph	Test specific changes
	start_of_sim	Logical start of simulation
Reset	reset	Reset DUT
cfg_dut	config_dut	Configuration of the DUT
start	start	Logical start of the test
	start_of_test	Physical start of test
wait_for_end	run	Body of test, end of test detection to be done here
stop	shutdown	Stop flow of stimulus
Cleanup	Cleanup	Let DUT drain and read final DUT state
Report	Report	Pass/Fail report (executed by each test)
	Final	Final checks and actions before simulation termination

Explicit Transactor Phasing, transactors begin to execute when the environment explicitly calls **vmm\_xactor :: start\_xactor** to start the Transactor. This then starts the **vmm\_xactor :: main** thread.

Eg: **Extension of vmm\_xactor :: main() task**

```

task mii_mac_layer :: main();
  fork
    super.main ();
  join_none
  ...
endtask: main

```



## Eg: Modeling Transactor

```
class my_vip extends vmm_xactor;
  `vmm_typename(my_vip)
  Function new(string name = "", vmm_object parent = null);
    super.new("vip", name);
    super.set_parent_object(parent);
  endfunction
  virtual function void start_xactor();
    super.start_xactor ();
    `vmm_note(log, "Starting...");
  endfunction
  virtual function void stop_xactor();
    super.stop_xactor ();
    `vmm_note(log, "Stopping...");
  endfunction
  `vmm_class_factory(my_vip)
endclass
```

## 4.1 Explicit Phasing

### Eg: Creation of Explicitly Phased Environment

```
class my_env extends vmm_env;
  `vmm_typename(my_env)
  my_subenv subenv1;
  my_subenv subenv2;

  function new ();
    super.new("env");
  endfunction

  virtual function void build();
    super.build ();
    this.subenv1 = new ("subenv1", this);
    this.subenv2 = new ("subenv2", this);
  endfunction

  virtual task start ();
    super.start();
    `vmm_note(log, "Started...");
    this.subenv1.start();
    this.subenv2.start();
  endtask

  virtual task wait_for_end ();
```

```

        super.wait_for_end();
        `vmm_note(log, "Running...");
        #100;
    endtask

    virtual task stop ();
        super.stop();
        `vmm_note(log, "Stopped...");
        this.subenv1.stop();
        this.subenv2.stop();
    endtask
endclass

```

**Note:** ‘*Super*’ keyword is being used to enable calling the methods from the base class instead of derived class with same method names

Eg: Creation of Explicitly Phased Test using above Environment

```

        `vmm_test_begin (test, my_env, "Test")
            env.run();
        `vmm_test_end(test)

```

Eg: Top Program to use the above test and environment

```

    program top;
        initial
        begin
            my_env env = new;
            vmm_test_registry::run(env);
        end
    endprogram

```

## 4.2 Implicit Phasing

In the implicit phasing execution model, transactors are self-controlled through built-in phasing mechanism. The environment automatically calls the phase specific methods in a top down, bottom up and forked fashion.

Implicit phasing works only with transactors that you base on the *vmm\_group* or *vmm\_xactor* class. The two use models are,

- If you want to call your Transactor phases from the environment, you should instantiate your *vmm\_xactor*(s) in *vmm\_env* or *vmm\_subenv*.
- If you want to have the environment implicitly calling Transactor phases, you should instantiate your *vmm\_xactor*(s) in *vmm\_group*.

Implicit phasing works only with classes that you base on the *vmm\_group* class.

### ***Eg: Creation of Implicitly Phased Sub-Environment***

```
class my_subenv extends vmm_group;
    \vmm_typename(my_subenv)
    my_vip vip1;
    my_vip vip2;
    function new(string name = "", vmm_object parent = null);
        super.new("vip", name, null);
        super.set_parent_object(parent);
    endfunction
    virtual function void build_ph();
        super.build_ph();
        this.vip1 = new("vip1", this);
        this.vip2 = new("vip2", this);
    endfunction
    virtual task start_ph();
        super.start_ph();
        \vmm_note(log, "Started...");
    endtask
endclass
```

### ***Eg: Creation of Implicitly Phased Environment***

```
class my_env extends vmm_group;
    \vmm_typename(my_env)
    my_subenv subenv1;
    my_subenv subenv2;
    function new();
        super.new("env");
    endfunction
    virtual function void build_ph();
        super.build_ph();
        this.subenv1 = new("subenv1", this);
        this.subenv2 = new("subenv2", this);
    endfunction
    virtual task start_ph();
        super.start_ph();
        \vmm_note(log, "Started...");
    endtask
    virtual task run_ph();
        super.run_ph();
        \vmm_note(log, "Running...");
        #100;
    endtask
    virtual task shutdown_ph();
```

```

        super.shutdown_ph();
        `vmm_note(log, "Stopped...");
    endtask
endclass

```

***Eg: Creation of Implicitly Phased Test using above sub\_env and env***

```

class test extends vmm_test;
    function new();
        super.new("Test");
    endfunction
    virtual task start_ph();
        super.start_ph();
        `vmm_note(log, "Started...");
    endtask
    virtual task shutdown_ph();
        super.shutdown_ph();
        `vmm_note(log, "Stopped...");
    endtask
endclass

```

Transactors are started during specific phases (not necessarily at start), run during a certain number of phases. Environment can suspend their execution thread and resume it and might stop it during another phase.

The ***vmm\_xactor*** class is the base class for transactors. It provides thread management utilities (***start, stop, reset\_xactor, wait\_if\_stopped***) that are not present in the other base classes. The ***vmm\_xactor*** offers both thread management and phase methods. It is important to understand to properly model transactors and how you model different behaviors at different phases. The simplest form for a Transactor is one whose behavior does not change between simulation phases. If you instantiate this Transactor in an implicitly phased environment, then it gets started by default.

## 5. Transactor Callbacks

Callback methods to monitor the data flowing through a Transactor to check for correctness inject errors or collect functional coverage metrics. You can adapt the Transactor callbacks to the needs of testcase or environment. Callback should be registered in the ***vmm\_xactor*** base class. However, calling the registered callback extensions is the responsibility of the Transactor extended from the base class.

### ***Transactor Callbacks Usage***

- Create a callback class with empty virtual methods
- Each virtual method represents an important stage of Transactor.
- The arguments of the virtual methods should contain necessary information that can be shared with the subscribers.

***Eg 1 :*** *class cpu\_driver\_callbacks extends vmm\_xactor\_callback;*

```
virtual task pre_trans (cpu_driver driver, cpu_trans tr, ref bit drop);  
endtask  
  
virtual task post_trans (cpu_driver driver, cpu_trans tr);  
endtask  
  
endclass
```

- At every important stage in the Transactor, call the corresponding method declared above through ***`vmm\_callback*** macro.

***Eg 2:*** *class cpu\_driver extends vmm\_xactor;*

```
virtual protected task main();  
  
    super.main();  
    ....  
    `vmm_callback(cpu_driver_callbacks, pre_trans(this, tr, drop));  
        if (tr.kind == cpu_trans::WRITE) begin  
            write_op(tr);  
        end  
        if (tr.kind == cpu_trans::READ) begin  
            read_op(tr);  
        end
```

```

`vmm_callback(cpu_driver_callbacks, post_trans(this, tr));
endtask

endclass

```

- A subscriber extends the callback class, fill the necessary empty virtual methods.

**Eg 3: class *cpu\_sb\_callback* extends *cpu\_driver\_callbacks*;**

```

    cntrlr_scoreboard sb;
    function new(cntrlr_scoreboard sb);
        this.sb = sb;
    endfunction

    virtual task pre_trans(cpu_driver drv, cpu_trans tr, ref bit drop);
        sb.cpu_trans_started(tr);
    endtask

    virtual task post_trans(cpu_driver drv, cpu_trans tr);
        sb.cpu_trans_ended(tr);
    endtask
endclass

```

- Register the subscriber callback class using method ***vmm\_xactor::append\_callback***.
- Then every time Transactor hits the defined important stages, subscriber methods will be called. Note that any number of subscribers with their own definition of virtual methods can get registered to a Transactor.

**Eg 4: class *cntrlr\_env* extends *vmm\_group*;**

```

    cpu_driver drv;
    virtual function void connect_ph();
        cpu_sb_callback cpu_sb_cbk = new(sb);
        cpu_cov_callback cpu_cov_cbk = new(cov);
        drv.append_callback(cpu_sb_cbk);
        drv.append_callback(cpu_cov_cbk);
    endfunction
endclass

```

## 6. Scenario Generators

- VMM provides a scenario macro to build most of the scenario code
- Customize for your own needs by adding own constraints and scenario selection rules

**Eg:** ``vmm_scenario_gen (transaction_class, scn_gen_name)`

```

`vmm_scenario_gen (atm_cell, "ATM Scn Gen")

```

## 1. Scenario Generator mainly contains

- 1 or more scenario classes
- Scenario selection class

Scenario Class will be derived from the transaction\_scenario class and the scenario base class is defined by the macro.

## 2. Scenario Class contains

- one or more scenarios selected by *scenario\_kind*
- Constraints to direct each scenario. Each scenario must be constrained with length and counter
- **apply()** task - send the scenario to an output channel

### Eg: Creating a Scenario

```
class my_scenario extends atm_cell_scenario
```

```
    `vmm_typename(my_scenario)
    int unsigned INC_VPI;                // Scenario kind must be defined
    constrained inc_vpi_scenario {
        ($void(scenario_kind) == INC_VPI) -> {
            length == 5;
            repeated == 0;
            foreach (items[i])
                if (i > 0) items[i].vpi == items[i-1].vpi+1;
        }
    }
    function new ();
        this.INC_VPI = define_scenario ("Inc VPI", 5);
    ...
    endfunction
...

    `vmm_class_factory(my_scenario)
endclass
```



**Scenario  
Constraint**

## Registering Scenarios

### 1. Adding Scenario Objects to Scenario Set

**Eg:** my\_scenario my\_scn = new( );  
env.scn\_gen.scenario\_set.push\_back(my\_scn); **//Add a new scenario**

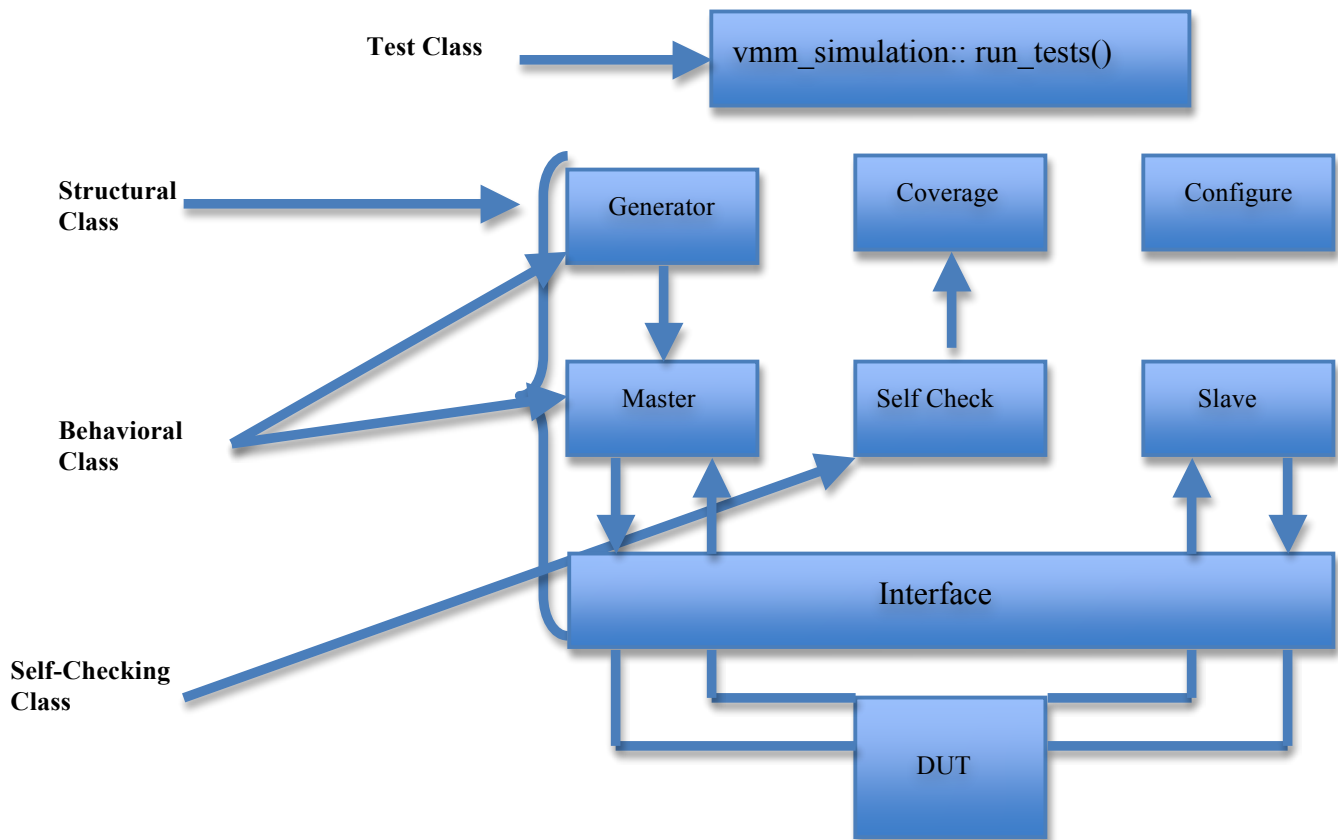
**Eg:** my\_scenario my\_scn = new( );  
env.scn\_gen.scenario\_set[0] = my\_scn; **// Replacing default atomic scenario**

## 2. Multistream Scenario

Eg: *task my\_ms\_scenario :: execute ( ref int n );*

```
vmm_channel to_ahb = get_channel ("AHB");  
vmm_channel to_eth = get_channel ("ETH");  
if (!this.ahb.randomize() )  
    `vmm_fatal ( log, "ahb randomization failed");  
do begin  
    to_ahb.put (this.ahb);  
end while ( this.ahb.status != ahb_cycle :: IS_OK );  
to_eth.put(this.eth);  
n++;  
endtask
```

## 7. VMM Testbench Essential Elements



(i) Test Class: vmm\_test

(ii) Structural Class: vmm\_group

(iii) Behavioral Class: vmm\_xactor

(iv) Communication Class: vmm\_channel (Eg: communication between Generator and Master)



There are three general categories of Transactors. Master, Slave and Monitors. All of these will be built by extending from the `vmm_xactor` base class.

Below are the examples of different types of transactors.

### Creating Transaction Classes

```
class Packet extends vmm_data;
  rand bit [3:0] sa, da;
  rand bit [7:0] payload[$];
  int active_sa[$], active_da[$];
  constraint valid {
    payload.size inside { [1:1024] };
    sa inside active_sa; da inside active_da;
  }
  `vmm_data_member_begin(Packet)
  `vmm_data_member_scalar(sa, DO_ALL)
  `vmm_data_member_scalar(da, DO_ALL)
  `vmm_data_memeber_scalar_array(payload, DO_ALL)
  `vmm_data__member_scalar_array(active_sa, DO_COPY)
  `vmm_data_member_scalar_array(active_da, DO_COPY)
  `vmm_data_member_end(Packet)
endclass
```

All transactions extend  
from `vmm_data`

### Creating Master Transactor Class

```
class master extends vmm_xactor;
  vmm_channel_typed # (Packet) in_chan;

  function new (string inst, int stream_id, vmm_object parent);
    super.new ("master", inst, stream_id, parent);
  end function

  protected task main ();
    super.main();
    forever begin
      Packet tr;
      this.wait_if_stopped_or_empty (this.in_chan);
      this.in_chan.activate(tr);
      this.send(tr);
      this.in_chan.remove();
    end
  endtask
endclass
```

All transactors extend  
from `vmm_xactor`

A typical Master Transactor waits for a transaction (typically passed in via a typed channel) to process.

## Creating Slave Transactor Class

```
class slave extends vmm_xactor;
  function new (string inst, int stream_id, vmm_object parent);
    super.new ("slave", inst, stream_id, parent);
  endfunction

  protected task main() ;
    super.main ();
    forever begin
      Packet tr;
      this.recv(tr);
    end
  endtask
endclass
```

**Note:** Slave transactors are infinite loops in main () emulating capture flop.

A slave Transactor class calls a physical layer device driver method to actively handshake with the DUT and reconstructs the transaction received from the DUT. If there is a passive monitor built in the testbench then the reconstructed transaction should be discarded. If no passive monitor exists on the interface then the reconstructed transaction should be passed to a scoreboard via either a channel or callback mechanism.

## Creating Monitor Transactor Class

```
class imonitor extends vmm_xactor;
  function new (string inst, int stream_id, vmm_object parent);
    super.new ("monitor", inst, stream_id, parent);
  endfunction

  protected task main();
    super.main();
    forever begin
      Packet tr;
      this.get_input_packet(tr);
      this.inp_vmm_sb_ds(tr);
    end
  endtask
endclass
```

```

class omonitor extends vmm_xactor;
  function new (...);
    super.new(...);
  endfunction

  protected task main();
    super.main();
    forever begin
      Packet tr;
      this.get_output_packet (tr);
      this.exp_vmm_sb_ds (tr);
    end
  endtask

```

Passive Monitors are different from the Master and Slave transactors in that they only observe interface handshakes. They do not actively participate in the interface handshake. Passive monitors exist for two primary reasons: protocol check and reconstruction of transaction to be passed on to a scoreboard.

The built-in methods, `inp_vmm_sb_ds ()` and `exp_vmm_sb_ds ()`, can be used to check the input transaction against the expected transaction.

## Creating Scoreboard Class

Data stream scoreboard **extends from vmm\_sb\_ds**. The `vmm_bs_ds` base scoreboard class has compare methods built-in. It can be extended to incorporate additional user specific requirements.

```

class scoreboard extends vmm_sb_ds;
  Packet pt;
  covergroup sb_cov;
    sa : coverpoint pkt.sa;
    da : coverpoint pkt.da;
    cross sa, da;
  endgroup
  function new (string name = "scoreboard");
    super.new(name);
    sb_cov = new;
  endfunction
  function bit compare ( vmm_data actual, vmm_data expected )
    if ( super.compare(actual, expected) ) begin
      $cast (this.pkt, actual);
      this.sb_conv.sample();
      return;
    end
    return 0;
  endfunction
endclass

```

## Creating Test Configuration Class

```
class test_cfg;
    rand int run_for_n_packets;
    rand int num_of_iports, num_of_oports;
    rand bit iport[16], oport[16];
    constraint valid {
        run_for_n_packets inside {[1:1000]};
        num_of_iports inside {[1:16]};
        num_of_oports inside {[1:16]};
        iport.sum() == num_of_iports;
        oport.sum() == num_of_oports;
    }
    virtual function void display ();
endclass
```

The configuration class is intended to define the parameters for a given testcase.

## Construct Testbench components

```
class tb_env extends vmm_group
    vmm_atomic_gen #(Packet) gen;
    master dvr;    slave rcv;
    imonitor imon;    omonitor omon;
    scoreboard sb;    test_cfg cfg;
    function new(string inst="", vmm_object parent);
        super.new("tb_env", inst, parent);
    endfunction
    function void build_ph();
        gen = new("gen", 0, null, this);
        sb = new("sb");
        dvr = new("dvr", 0, this);
        rcv = new("rcv", 0, this);
        imon = new("imon", 0, this);
        omon = new("omon", 0, this);
        cfg = new();
    endfunction
    .....
endclass
```

## Configure and Connect Transactors

1. Connect the components in connect\_ph()
2. Set test controls in start\_of\_sim\_ph()
3. Wait for consensus in run\_ph()

## Run-Time Seeds Create Different Tests

The ability to set random seed at run-time with *+ntb\_random\_seed* option is very useful for creating multiple reproduce-able tests with one *simv*.

But user must pick and set the seed. An alternative can be *+ntv\_random\_seed\_automatic*. With this option, the seed is randomly picked by *vcs*. Every run of the same *simv* binary will result in running simulation with a different seed. When using the run-time option, one must retrieve and store the seed being used with *\$get\_initial\_random\_seed()*.

The *vmm\_env* base class automatically calls *\$get\_initial\_random\_seed()* and displays the random seed for you.

The default seed if neither option is applied is 1.

## Develop a collection of Tests

```
class all_ports extends vmm_test;
  function new(string name, doc);
    super.new (name, doc);
  endfunction
  function void configure_test_ph();
    env.cfg.num_of_iports = 16;
    env.cfg.num_of_oports = 16;
    env.cfg.num_of_iports.rand_mode(0);
    env.cfg.num_of_opors.rand_mode(0);
  endfunction
endclass
all_ports test_all = new("all_ports", "Testing all ports");
```

```
class ten_packets extends vmm_test;
  function new(string name, doc);
    super.new(name, doc);
  endfunction
  function void configure_test_ph();
    env.cfg.num_of_iports = 10;
    env.cfg.num_of_oports = 10;
    env.cfg.run_for_n_packets.rand_mode(0);
  endfunction
endclass
ten_packets test_ten_packets = new("ten_packets", "Small Test");
```

## Execution of Tests:

### - At test top level:

```
program automatic test;
  `include "tb_env.sv"
  `include "tests.inc"
  tb_env env = new();
  initial begin
    vmm_simulation :: list();
    vmm_simulation :: run_tests();
  end
endprogram
```

#### - simv command line:

- ./simv +vmm\_test = all\_ports (These are name of the tests specified)
- ./simv +vmm\_test = ten\_packets
- ./simv +vmm\_test = ALL\_TESTS

## VMM Generators

VMM has two types of generators. Atomic generator and Scenario generator.

Atomic generator is a simple generator, which generates transactions randomly.

``vmm_atomic_gen` is a macro which is used to define a class named

`<class_name>_atomic_gen` for any user-specified class derived from `vmm_data`, using a process similar to the ``vmm_channel` macro.

To use `<class_name>_atomic_gen` class, a `<class_name>_channel` must exist.

`<class_name>_atomic_gen` generates transactions and pushes it to `<class_name>_channel`. A

`<class_name>_channel` object can be passed to generator while constructing.

Lets create atomic generator for `auto_packet` class in file `auto_packet.sv`

1. define ``vmm_atomic_gen` macro for packet class. This macro creates a `packet_atomic_gen` class creates and randomizes packet transactions.

```
`vmm_atomic_gen(auto_packet,"packet atomic generator")
```

2. define ``vmm_channel` for the packet class. This macro creates a `packet_channel`, which will be used by the `packet_atomic_gen` to store the transactions. Any other component can take the transactions from this channel.

```
`vmm_channel(auto_packet)
```

3. Create an object of `packet_atomic_gen`.

```
packet_atomic_gen pkt_gen = new ("Atomic Gen","test");
```

4. Set the number of transactions to be generated to 4

```
pkt_gen.stop_after_n_insts = 4;
```

5. Start the generator to generate transactions. These transactions are available to access through pkt\_chan as soon as they are generated.

```
pkt_gen.start_xactor();
```

6. Collect the packets from the pkt\_chan and display the packet content to terminal.

```
pkt_gen.out_chan.get(pkt);
```

```
pkt.display();
```

### Factory Class Needs

- Factories are blue print classes that generate objects of a specific kind
- A factory should be able to generate objects of the specified class
- Factories should also be able to generate customized transactions if specified at test case level

Eg: Allocate new objects using a blueprint instance, via a virtual method

```
class bfm extends vmm_xactor;
  transaction blueprint;
  ....
  task build_ph();
    blueprint = new ();
  endtask
  protected task main();
    super.main();
    forever begin
      transaction tr;
      tr = blueprint.allocate ();
      ...
      process(tr);
    end
  endtask
endclass
```

```
class transaction extends vmm_data;
  ....
  virtual function vmm_data allocate;
    transaction txn = new();
    return txn;
  endfunction
endclass
```

In above example for a factory design, we allocate from a single instance, via a virtual method, that is then stored in the local variable instead of allocating into local variable directly.

In Factory class, we can replace a transaction by a derived class or we can replace a scenario by another scenario.

## Two ways of overriding factory

[i] **by copy()** : replace existing factory by an object of same/derived instance with the values of the instance copied.

[ii] **by new ()** : replace existing factory by a new derived object.

User doesn't have to necessarily extend `vmm_object` or `vmm_data` to make use of factory service.

## Scoreboard

- Self-checking testbenches need scoreboards.
- Reusable scoreboards can be used for multiple testbenches.
- Proper understanding of the DUT is necessary to design an efficient scoreboard.
- Different score boarding mechanisms are used for different applications.

## Creating & Accessing Scoreboards

- Use `vmm_sb_ds` macro
- If the DUT behavior is single stream and no transformation, no extension to base class is required.

## Syntax

```
//Extend vmm_sb_ds
class my_sb extends vmm_sb_ds;
.....
endclass
```

## Eg: Use `vmm_xactor`'s scoreboard methods

```
class bus_master extends vmm_xactor;
.....
  `vmm_callback(bus_master_cb, post_tr(this, tr);
  this.inp_vmm_sb_ds(tr); // Add input packet
endclass

class bus_mon extends vmm_xactor;
.....
  `vmm_callback (bus_mon_cb, post_tr (this, tr) ) ;
  this.exp_vmm_sb_ds (tr); // Check with expected packet
endclass
```

## Connecting Scoreboard

```
class my_env extends vmm_group;
  vmm_sb_ds sb;
  bus_master xtor_mstr;
  bus_mon xtor_mon;
```



```

virtual function build_ph();
    super.build ();
    this.sb = new(...); // Creating Scoreboard
    this.xtor_mstr = new (...);
    this.xtor_mon = new (...);
endfunction: build_ph

virtual function connect_ph()
    this.xtor_mstr.register_vmm_sb_ds (this.sb,
                                     vmm_sb_ds :: INPUT,
                                     vmm_sb_ds :: IN_ORDER) ;
    this.xtor_mon.register_vmm_sb_ds(this.sb,
                                     vmm_sb_ds :: EXPECT,
                                     vmm_sb_ds :: IN_ORDER);

    endfunction: connect_ph
endclass

```

### Scoreboard Compares

- In addition to vmm\_data :: compare() the scoreboard has two compare functions
- vmm\_sb\_ds :: quick\_compare ( )
  - Called by expect\_with\_losses ( ) & vmm\_sb\_ds::compare()
  - Minimal set of checks to identify unique matches
  - May require calling vmm\_data :: compare()
  - Default behavior returns 1
- vmm\_sb\_ds :: compare ( )
  - Normal scoreboard compare routine
  - Calls quick\_compare ( ) followed by vmm\_data :: compare ( )
- Overload either of these functions or custom compares

### Scoreboard - Transformation

- Specified via virtual method vmm\_ds\_sb :: transform ( )
- Transform can be one-to-one, one-to-many, many-to-one

**Eg:**

```

class ahb_to_ocp_sb extends vmm_sb_ds;
    virtual function bit transform ( input vmm_data in_pkt, output vmm_data out_pkts[] );
        ahb_tr in_tr;
        ocp_tr out_tr = new;
        $cast ( in_tr, in_pkt );

        //convert in_tr to out_tr. One-to-one transform
        out_tr = .....

        out_pkts = new [1];

```

```

        out_pkts[0] = out_tr; // Fill the out packets array.
    endfunction
endclass

```

### 6.3 Message Service

- Transactors. Scoreboards, assertions, environment and testcases use messages to report any definite or potential errors detected. They may also issue messages to indicate the progress of the simulation or provide additional processing information to help diagnose problems. A message service is only concerned with the formatting and issuance of messages, not their causes. For example, the time reported in a message is the time at which the message was issued, not the time a failed assertion started. The VMM message service uses the following concepts to describe and control messages.

1. **Message Source:** Message source can be any component of a testbench. Messages from each source can be controlled independently of the message from other sources.

2. **Message Filters:** Filters can prevent or allow a message from being issued. They are associated and disassociated with message sources. Message filters can promote or demote messages severities, modify message types and their simulation handling.

3. **Message Type:** Individual messages are categorized into different types to issue the message.

Eg: vmm\_log :: FAILURE\_TYP => An error has been detected.

vmm\_log :: NOTE\_TYP => Normal message used to indicate the simulation progress.

vmm\_log :: DEBUG\_TYP => Message used to provide additional information designed to help diagnose the cause of a problem.

vmm\_log :: TIMING\_TYP => Timing error has been detected.

4. **Message Severity:** Individual messages are categorized into different severities to issue the message. A message's severity indicates its importance and seriousness and must be chosen with care

### 6.4 Creating Tests

#### Purpose

- All tests will be defined as a class
- Supports explicit and implicit phasing with specific timelines
- Compile and elaborate once, run specific tests with run time selection
- Separate test and environment compilation guidelines

## Benefits

- All tests defined in program or modules
- Test automatically registered once declared
- No recompilation needed for each test

## Implicit phased VMM test

- Test comes with 3 timelines: pre\_test, top\_test, post\_test

## Overriding Options and Configurations

### 1. Possibility to replace environment factories and scenarios

- Done in top-test timeline, before physical test start
- Use vmm\_test :: configure\_test\_ph() to add testcase specific code

## Example Testcase

```
class test1 extends vmm_test;

  virtual function void configure_test_ph();

    vmm_opts :: set_int ("top.env.timeout", 10_000); // Set simulation timeout
    vmm_opts :: set_int ("top.msgen.stop_after_n_scenarios", 50); // No. of scenarios
    vmm_opts :: set_int ("top.cpu_chan.record", 1'b0); // Turn off channel recording

endfunction

endclass: test1
```

## Test Concatenation

- %simv +vmm\_test = test1 + test2
- %simv +vmm\_test = ALL\_TESTS

## Example to write tests in the top module

```
program top
  my_env env;
  test1 t1;
  test2 t2;
  test3 t3;
  initial begin
    env = new("env",.....);
    t1 = new ("test1");
    t2 = new ("test2");
    t3 = new ("test3");
    vmm_simulation :: run_tests ();
  end
endprogram
```

## 7. Hands-on VMM testbench template creation

### VMM Testbench template creation using *vmmgen* mode

1. Using quick mode to create the whole testbench template.
2. Creating individual testbench components one by one.

**Task 1:** Execute the below commands written in bold

**> vmmgen -q**

You will be prompted with the following questions one after another

- 1. Would you have sub environments in your environment? enter (y/n) :**
- 2. Would you be associating RAL models in your environment class? enter (y/n) :**
- 3. Would you be using RTL configurations in your environment? enter (y/n) :**

**Enter option ‘n’ for all the above questions.**

You will then be prompted with the following questions:

- 1. Enter the environment name:**

Specify name `cntrl_env`

For the next prompt:

- 2. Enter the name of transaction class:**

Specify name `cpu_trans`

- 3. Last prompt is: Would you like to have another transaction class? enter (y/n):**

Enter option “n” to complete template generation.

#### **Note:**

1. The above steps will create a number of VMM testbench template files for you. Browse through the files to see general guidelines of VMM testbench code. The environment file is in the **proj/cntrlr\_env/env** directory. The other source code files are in the **proj/cntrlr\_env/src** directory
2. Open the **cntrl\_env.sv** file (in **proj/cntrlr\_env/env**). Look through all the phases and add display statements within each phase as shown below.

```
task cntrl_env_env :: build_ph() ;  
    super.build_ph();  
    `vmm_note (log, "Entering build_ph().... ");  
    ....
```

```
endtask : build_ph
```

3. Run the simulation by going into **proj/cntrlr\_env/run** directory

4. Use **make** to compile and run the simulation and capture the log

> **make | tee log**

Take a look at the log file, you should see simulation is executing the sequence as shown in the tutorial.

**Task 2:** Creating individual testbench files using **vmmgen**.

1. Execute **vmmgen** command to build individual testbench components:

> **vmmgen**

You will be prompted with the following question:

**Which VMM version would you be using?**

1) **VMM-1.1**

2) **VMM-1.2**

**Select [1-2] [Default: 2]:**

Specify option 2

Then you will see prompt:

1) **Enter 1 to Create Complete Environment**

2) **Enter 2 to Generate Individual Template**

Again specify option 2

3) You will be prompted with: **Which template do you wish to generate?**

Select option 1, to create a VMM transaction class.

You will then be prompted with the following question:

**Do you want to create your own methods [Instead of vmm shorthand macros]?**

**Select [y/Y/n/N] [Default: n]:**

Enter option n

4. The last question to create transaction class is:

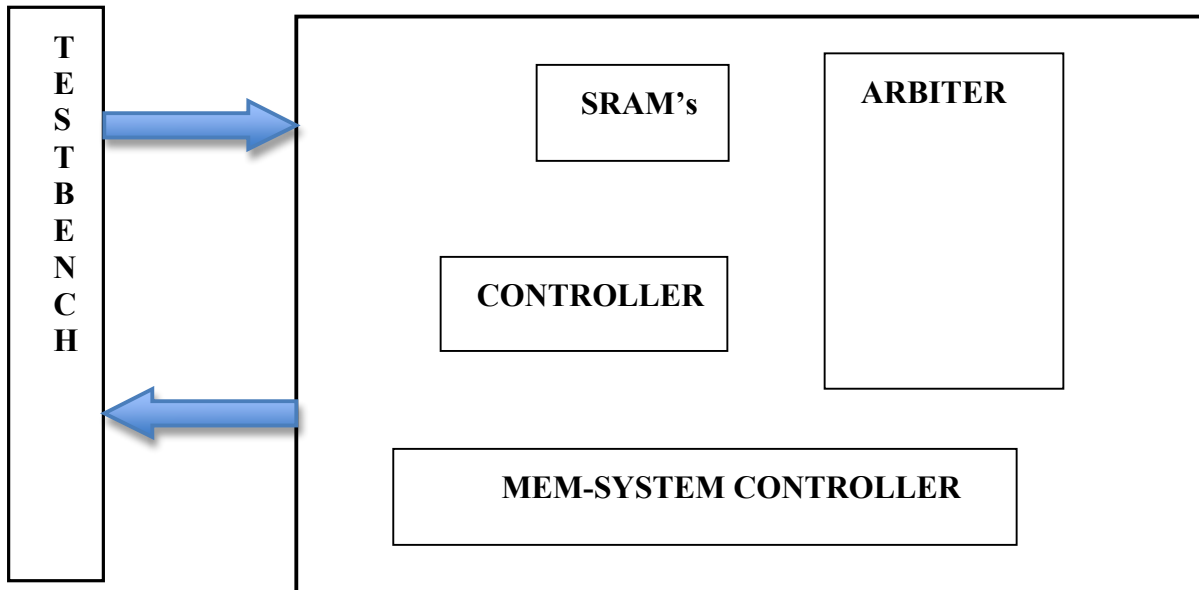
**Name of transaction descriptor class? :**

Specify name **cpu\_trans**

A **cpu\_trans.sv** is created in the directory. This creates a file with vmm transaction class template.

You can even try to create other individual testbench components. It helps to minimize the amount of typos that would otherwise occur if you were to type all codes manually.

## 8. Example DUT Details



A Memory Controller block is the Design Under Test.

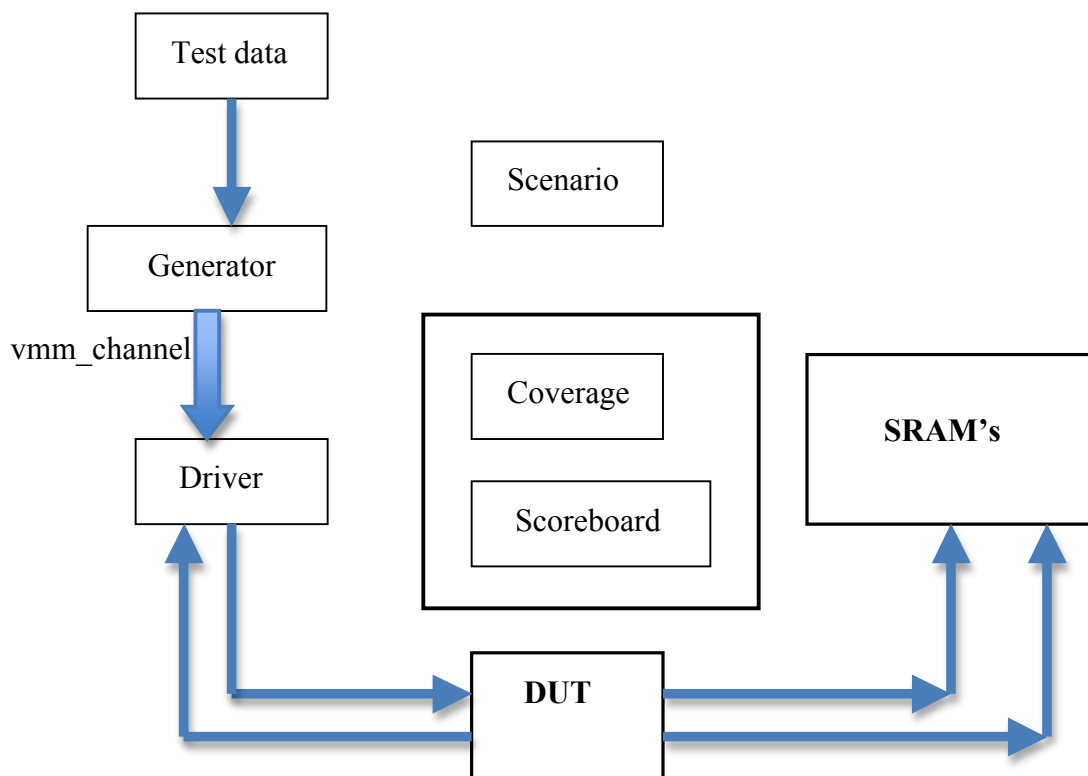
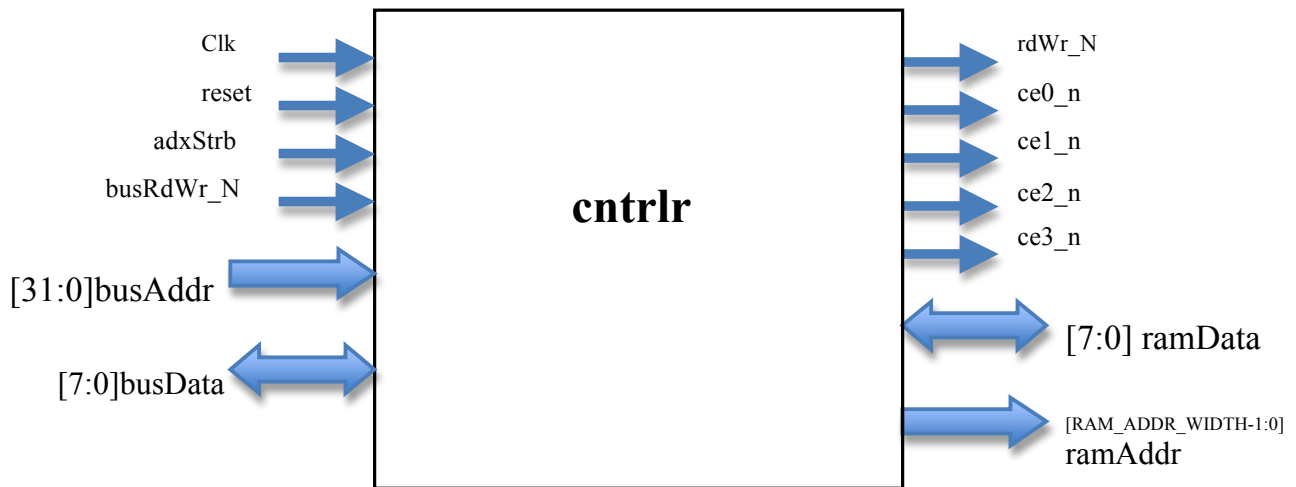
1. Has a master interface (CPU) and a slave interface (SRAM) with multiple device selects.
2. 1,2 or 4 SRAM's can be connected to the DUT.
3. The size of each SRAM device can be configured (256, 512, 1024)
4. Memory controller performs read and write operation on to the SRAM memory.
  - Master interface (cpu) instantiates the read or write operation.

Slave interface (sram) responds to the read/write operations.

The DUT files are located in the below format and inside "*hdl*" folder

1. arb.v // Arbiter verilog design file
2. cntrlr.v // Controller verilog design file
3. memsys.v // Memory system controller file
4. sram.v // SRAM design verilog file

## Controller DUT



Callbacks take place between **Driver** ⇔ **Coverage** ⇔ **SRAM's**

TLM ports between **Driver** ⇔ **Scoreboard** ⇔ **SRAM's**

**All the design blocks need to be tested, hence separate folders are created testing individual design blocks. All the folders contains interface, top level harness, coverage and transaction files.**

## Writing Scenarios

Different scenarios have been written to test the functionalities such as read/write or both through CPU onto the SRAM memory. We can find all the scenario files in the “**scenarios**” folder.

For example, below we can see scenario written to randomly create read scenarios. Pay attention on the syntax used to create scenarios.

```
class cpu_read_scenario extends cpu_rand_scenario;
  `vmm_typename(cpu_read_scenario)

  `vmm_scenario_new(cpu_read_scenario)

  `vmm_scenario_member_begin(cpu_read_scenario)
  `vmm_scenario_member_end(cpu_read_scenario)

  // Override the execute task to generate READ transactions
  virtual task execute(ref int n);
    cpu_trans tr;
    vmm_channel chan = get_channel("cpu_chan");
    $cast(tr, blueprint.copy());
    tr.stream_id = this.stream_id;
    tr.scenario_id = this.scenario_id;
    tr.data_id = 0;

    //Randomize transaction to generate READ transactions
    if (!tr.randomize() with { kind == READ; address == addr; })
      `vmm_fatal(log, "Read Scenario randomization Failed!");
    chan.put(tr);
    n++;
  endtask

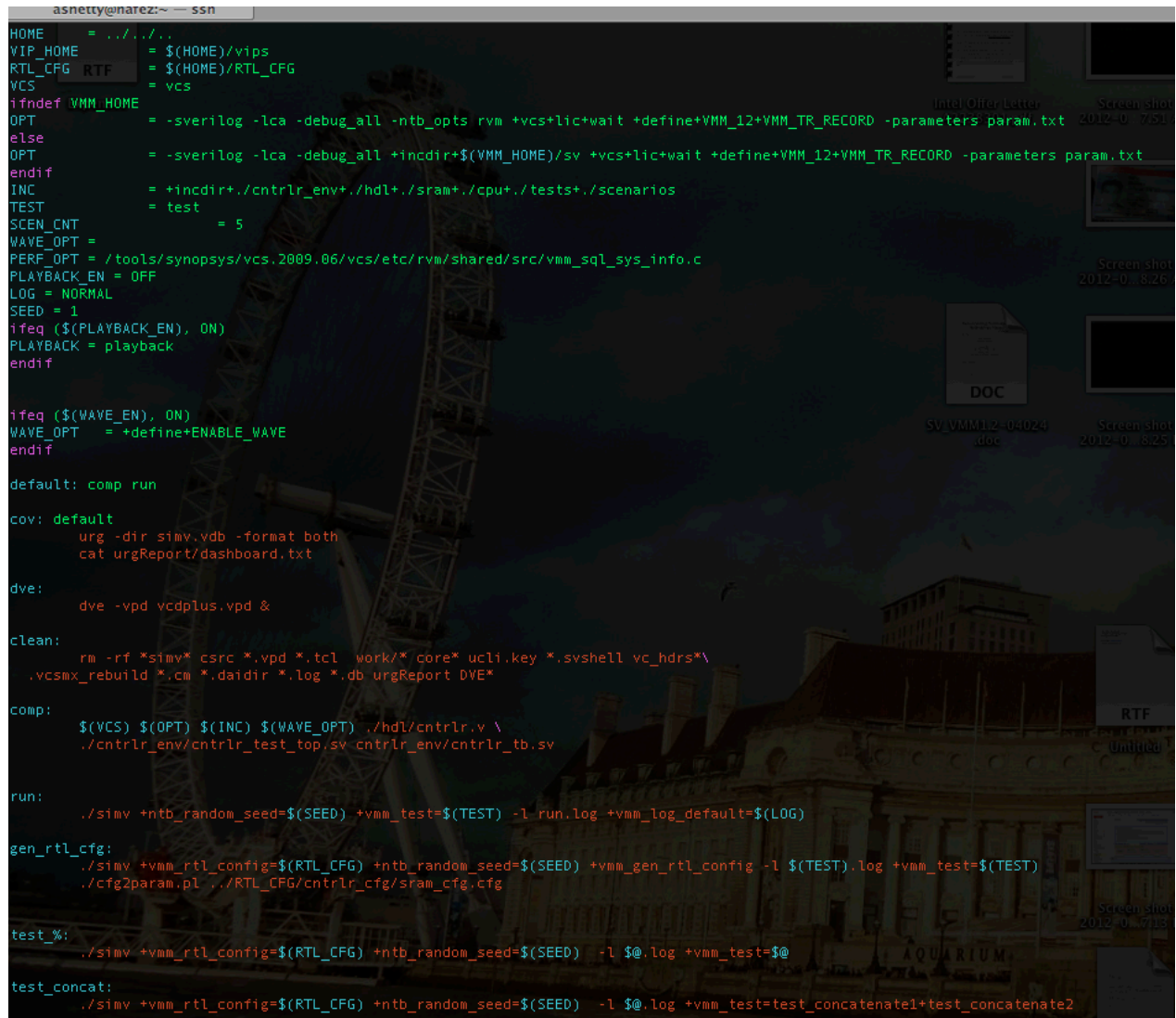
  `vmm_class_factory(cpu_read_scenario)
endclass
```



## Creation of Makefile

Makefile is being created to perform all the functions such as Compile, Run, DVE, Cleaning, Coverage and Selection of Tests. Makefile can be found at the below location

> **cd /packages/synopsys/setup/verification\_flow/VMM/solutions**



```
asnetty@narezi:~ - ssh
HOME = ../../..
VIP_HOME = $(HOME)/vips
RTL_CFG RTF = $(HOME)/RTL_CFG
VCS = vcs
ifndef VMM_HOME
OPT = -sverilog -lca -debug_all -ntb_opts rvm +vcs+lic+wait +define+VMM_12+VMM_TR_RECORD -parameters param.txt
else
OPT = -sverilog -lca -debug_all +incdir+$(VMM_HOME)/sv +vcs+lic+wait +define+VMM_12+VMM_TR_RECORD -parameters param.txt
endif
INC = +incdir+./cntrlr_env+./hdl+./sram+./cpu+./tests+./scenarios
TEST = test
SCEN_CNT = 5
WAVE_OPT =
PERF_OPT = /tools/synopsys/vcs.2009.06/vcs/etc/rvm/shared/src/vmm_sql_sys_info.c
PLAYBACK_EN = OFF
LOG = NORMAL
SEED = 1
ifeq ($(PLAYBACK_EN), ON)
PLAYBACK = playback
endif

ifeq ($(WAVE_EN), ON)
WAVE_OPT = +define+ENABLE_WAVE
endif

default: comp run

cov: default
    urg -dir simv.vdb -format both
    cat urgReport/dashboard.txt

dve:
    dve -vpd vcdplus.vpd &

clean:
    rm -rf *simv* csrc *.vpd *.tcl work/* core* ucli.key *.svshell vc_hdrs* \
    .vcsmx_rebuild *.cm *.daidir *.log *.db urgReport DVE*

comp:
    $(VCS) $(OPT) $(INC) $(WAVE_OPT) ./hdl/cntrlr.v \
    ./cntrlr_env/cntrlr_test_top.sv cntrlr_env/cntrlr_tb.sv

run:
    ./simv +ntb_random_seed=$(SEED) +vmm_test=$(TEST) -l run.log +vmm_log_default=$(LOG)

gen_rtl_cfg:
    ./simv +vmm_rtl_config=$(RTL_CFG) +ntb_random_seed=$(SEED) +vmm_gen_rtl_config -l $(TEST).log +vmm_test=$(TEST)
    ./cfg2param.pl ../RTL_CFG/cntrlr_cfg/sram_cfg.cfg

test_%:
    ./simv +vmm_rtl_config=$(RTL_CFG) +ntb_random_seed=$(SEED) -l $@.log +vmm_test=$@

test_concat:
    ./simv +vmm_rtl_config=$(RTL_CFG) +ntb_random_seed=$(SEED) -l $@.log +vmm_test=test_concatenate1+test_concatenate2
```

## Writing Testcases:

Multiple testcases are written to test the read/write features among the different design blocks and are available in the folder “**tests**”. All the individual tests include the respective scenarios to run the tests.

For example, lets look at cpu\_read testcase, which communicates with cpu and sram



## Coverage Model

> make cov

```
Test Case test Done
$finish at simulation time 43550
VCS Simulation Report
Time: 43550
CPU Time: 1.080 seconds; Data structure size: 0.1Mb
Mon Apr 30 14:26:58 2012
urg -dir simv.vdb -format both
URG Version D-2010.06 Copyright (c) 1991-2010 by Synopsys Inc.

Note-[URG-RDG] Report directory generated
Report written to directory urgReport

cat urgReport/dashboard.txt
Dashboard

Date: Mon Apr 30 14:26:59 2012

User: ashetty

Version: D-2010.06
Command line: urg -dir simv.vdb -format both

-----
Total Coverage Summary
SCORE GROUP
87.70 87.70
```

## Running Tests

> make test\_concat

```
+-----+-----+-----+-----+-----+-----+-----+
| CPU->SRAM | Insert | Matchd | MsMtch | Droppd | NotFnd | Orphan |
+-----+-----+-----+-----+-----+-----+-----+
| CPU->SRAM_0 | 0031 | 0031 | 0000 | 0000 | 0000 | 0000 |
| CPU->SRAM_1 | 0021 | 0021 | 0000 | 0000 | 0000 | 0000 |
| CPU->SRAM_2 | 0023 | 0023 | 0000 | 0000 | 0000 | 0000 |
| CPU->SRAM_3 | 0025 | 0025 | 0000 | 0000 | 0000 | 0000 |
+-----+-----+-----+-----+-----+-----+-----+
| TOTAL | 0100 | 0100 | 0000 | 0000 | 0000 | 0000 |
+-----+-----+-----+-----+-----+-----+

+-----+-----+-----+-----+-----+-----+-----+
| SRAM->CPU | Insert | Matchd | MsMtch | Droppd | NotFnd | Orphan |
+-----+-----+-----+-----+-----+-----+-----+
| SRAM_0->CPU | 0031 | 0031 | 0000 | 0000 | 0000 | 0000 |
| SRAM_1->CPU | 0021 | 0021 | 0000 | 0000 | 0000 | 0000 |
| SRAM_2->CPU | 0023 | 0023 | 0000 | 0000 | 0000 | 0000 |
| SRAM_3->CPU | 0025 | 0025 | 0000 | 0000 | 0000 | 0000 |
+-----+-----+-----+-----+-----+-----+-----+
| TOTAL | 0100 | 0100 | 0000 | 0000 | 0000 | 0000 |
+-----+-----+-----+-----+-----+-----+

Simulation PASSED on ./ (./) at 92050 (0 warnings, 0 demoted errors & 0 demoted warnings)
Normal[NOTE] on vmm_simulation(class) at 92050:
Test Case test_write Done

SUMMARY OF ALL TESTS
test_read : PASSED (0 Errors, 0 Warnings)
test_write : PASSED (0 Errors, 0 Warnings)
TOTAL = 0 Errors, 0 Warnings

$finish at simulation time 92050
VCS Simulation Report
```

## 10. Accessing example files from Hafez Server

[1] Copy the example files from the server using the below command into your present working directory

```
> cp -rpf /packages/synopsys/setup/verification_flow/VMM/.
```

You can also view your present working directory path by using the command

```
> pwd
```

Note: We can access all the design files inside **hdl** folder and all the verification components such as tests, scenarios in their respective folders.

## 11. References & Books

[1] “*Verification Methodology Manual for System Verilog*” by Janick Bergeron, Eduard Cerny, Alan Hunter and Andrew Nightingale.

[2] <http://www.vmmcentral.org/>

[3] <http://www.vmmcentral.org/vmartialarts/>