VMM PRIMER

Composing Environments

Author(s):

Janick Bergeron

Version 1.2 / March 18, 2008

Introduction

The VMM Environment Composition application package is a set of methodology guidelines and support classes to help create system-level verification environments from block-level environments.

This primer is designed to learn how to create a block-level environment that will be reusable in a system-level verification environment. Other primers cover other aspects of developing VMM-compliant verification assets, such as transactors, verification environments and the Register Abstraction Layer package.

This primer assumes that you are familiar with VMM. If you need to ramp-up on VMM itself, you should read the other primers in this series.

The design used to illustrate how to use the VMM Environment Composition application package is the OpenCore Ethernet Media Access Controller (MAC). Despite being a fairly complex design, this primer focuses on a single aspect of its functionality: the verification of its transmission path. This design was chosen because it exhibits all of the challenges in reusing parts of its verification environment: It is configurable and it requires access to a shared resource. The system-level design will be a simple composition of two MAC blocks.

A complete verification environment that can verify the entire functionality of this design requires many more elements and additional capabilities.

This document is written in the same order you would create a verification environment with portions designed to be reused in a system-level verification environment. As such, you should read it in a sequential fashion. You can use the same sequence to create your own reusable sub-environments.

Step 1: Planning

Creating block-level environments that can be reused in a system-level environment does not happen by accident. They have to be structured and architected to make then reusable.

First, the entire block-level environment may not be reusable. Interfaces that are visible and/or controllable at the block level may no longer be available at the system level. Also, low-level random stimulus at the block level may need to carry well-formed higher-level information at the system-level. It is important to identify which portion(s) of the block-level environment that would be reusable in a system-level context.

Figure 1 shows the structure of the verification environment surrounding the Ethernet MAC block. The firmware emulation transactor configures and controls the registers inside the design through a RAL model. The DMA memory, not being

mapped in the DUT address space, is not included in the RAL model. The firmware emulation transactor accesses the DMA memory directly through backdoor read()/write() methods provided by the RAM model itself. The detailed operation of the firmware emulation layer to successfully transmit a frame is described in the Memory Allocation Manager Primer.

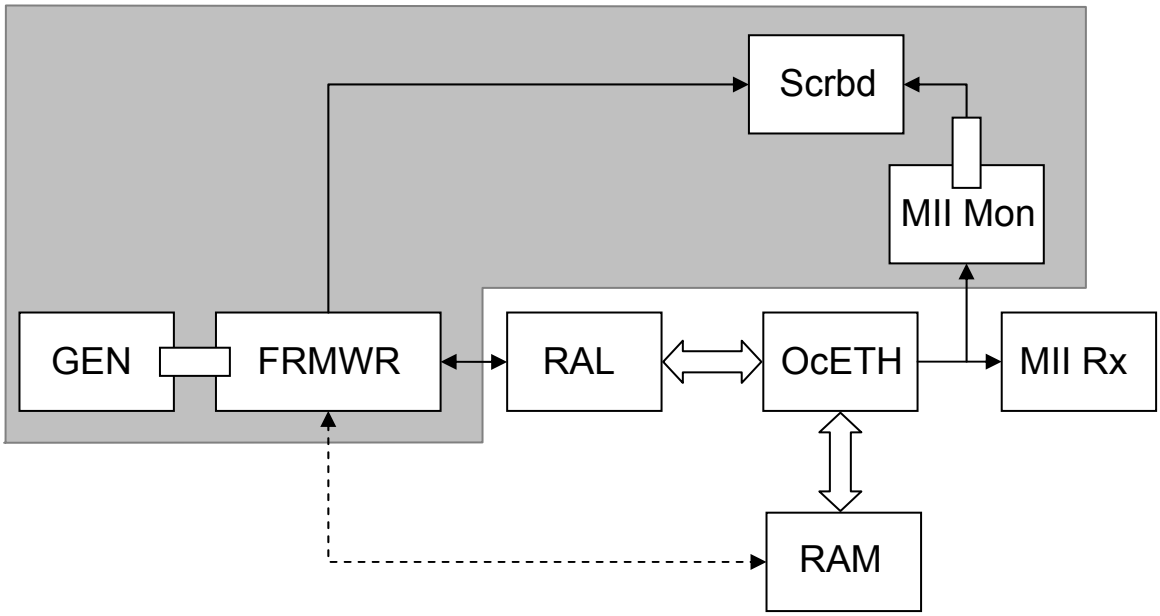

Figure 1: Block-Level Verification Environment

The shaded portion of the block-level environment will be made reusable in a system-level environment.

A passive monitor is used to extract information from the MII interface. The same information could have been extracted from the active MII receiver transactor. However, the MII interface may not be externally visible at the system-level and thus not in need of a receiver to properly terminate it. Using a monitor allows the self-checking portion of the block-level environment to be reused in an environment where that interface is internal.

The generator that provides stimulus to the firmware emulation block will be made optional. This will enable the firmware layer to be fed from a high-level protocol stimulus source, should the need arise.

The Register Abstraction Layer is not included in the reusable portion because the address and means of accessing the registers will likely be different between the block-level environment and the system-level environment. It will be up to the system-level environment to provide a reference to a suitable RAL model for the firmware emulation layer to be able to properly configure and control the corresponding block.

The system-level environment used in this primer is shown in Figure 2. The reusable sub-environment from Figure 1 is used twice with slightly different configurations.

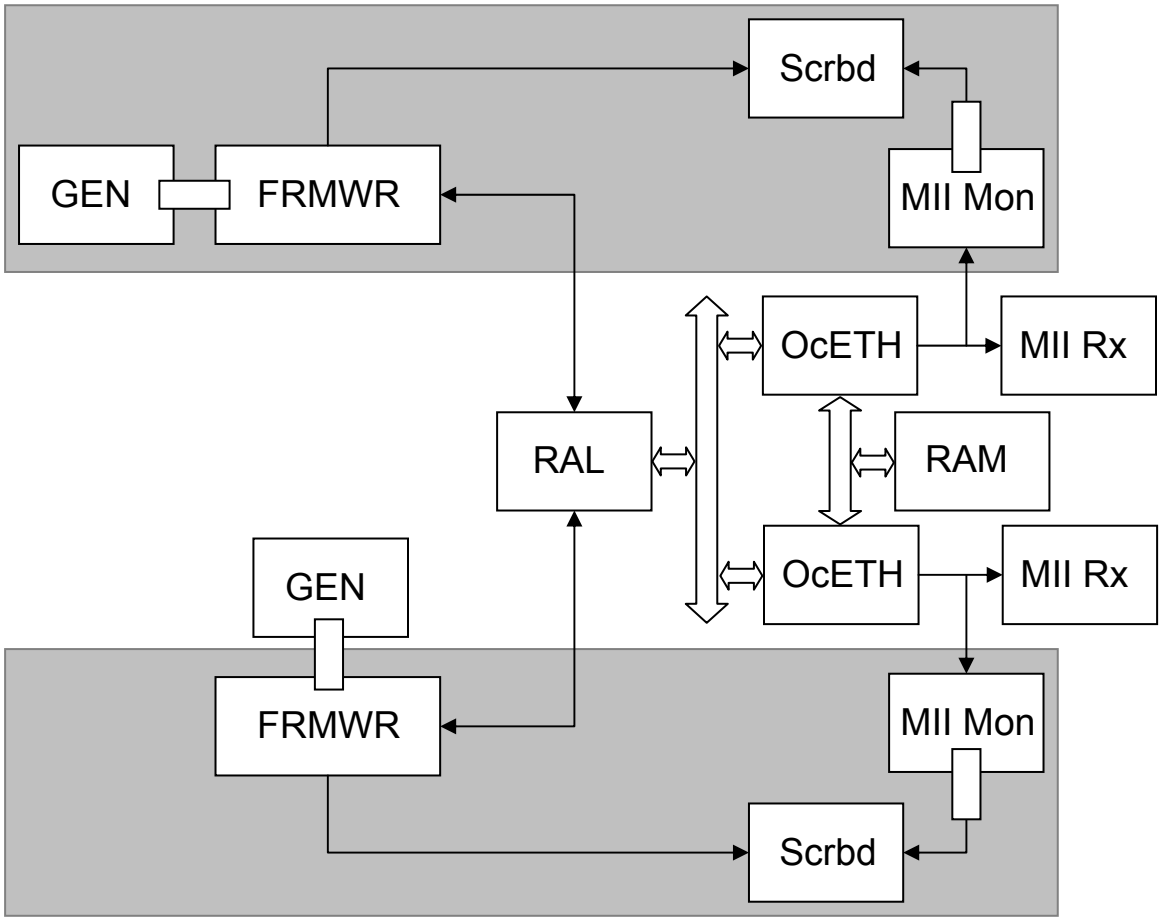

Figure 2: System-Level Verification Environment

Step 2: Encapsulation

The portion of the block level environment that will be reusable, called a sub-environment, is encapsulated in a class extended from the vmm_subenv base class. The constructor arguments must include:

- An instance name.
- A sub-environment configuration descriptor.
- A vmm_consensus reference.
- All virtual interfaces and channels that cross the sub-environment boundary.

SubEnv/eth_subenv.sv

```
class oc_eth_subenv_cfg;
   ...
endclass: oc_eth_env_cfg

class oc_eth_env extends vmm_subenv;
   ...
   function new(string                  inst,
                oc_eth_env_cfg          cfg,
                vmm_consensus           end_test,
                virtual mii_if.passive  mii_tx,
                eth_frame_channel       tx_chan,
                ...);
      super.new(”OcEth SubEnv”, inst, end_test);
      ...
   endfunction: new
   ...
endclass: oc_eth_subenv
```

The reference to the “tx_chan” argument is optional. If set to *null*, it will be assumed to not cross the sub-environment boundary and the generator will be internally allocated and connected. If it is not *null*, the reference will be used as-is and assumed to be properly fed from some source external to the sub-environment.

SubEnv/eth_subenv.sv

```
class oc_eth_subenv_cfg;
   ...
endclass: oc_eth_subenv_cfg

class oc_eth_subenv extends vmm_subenv;
   ...
   eth_frame_atomic_gen src;
   ...
   function new(string                  inst,
                oc_eth_env_cfg          cfg,
                vmm_consensus           end_test,
                virtual mii_if.passive  mii_tx,
                eth_frame_channel       tx_chan,
                ...);
      super.new(”OcEth SubEnv”, inst, end_test);
      ...
      if (tx_chan == null) begin
         this.src = new(“OcEth SubEnv Src”, inst);
         tx_chan = this.src.out_chan;
         ...
      end
      ...
   endfunction: new
   ...
endclass: oc_eth_subenv
```

The sub-environment has additional logical interfaces required to properly interact with other components of the overall environment. These logical interfaces must also be provided as constructor arguments and include:

- A reference to a RAL model for the block. This will enable the firmware emulator to configure the block regardless of the physical interface or base address it will be located under.
- A reference to system memory for backdoor read/write. If the system memory were mapped into the space of the block, it could be accessed through the RAL model for that block. But such is not the case here.
- A reference to the memory allocation manager for the system memory. Since the system memory is a potentially shared resource, it my be managed by a shared resource manager.

SubEnv/eth_subenv.sv

```
class oc_eth_subenv extends vmm_subenv;
   ...
   eth_frame_atomic_gen src;
   ...
   function new(string                  inst,
                oc_eth_env_cfg          cfg,
                vmm_consensus           end_test,
                virtual mii_if.passive  mii_tx,
                eth_frame_channel       tx_chan,
                ral_block_oc_ethernet   ral,
                wb_ram                  dma_ram,
                vmm_mam                 dma_mam);
      super.new("OcEth SubEnv", inst, end_test);
      ...
      if (tx_chan == null) begin
         this.src = new("OcEth SubEnv Src", inst);
         tx_chan = this.src.out_chan;
         ...
      end
      ...
   endfunction: new
   ...
endclass: oc_eth_subenv
```

The subenvironment is then instantiated in the block-level environment and constructed, like any other transactor, in the build() step.

SubEnv/blk_env.sv

```
class test_cfg;
   rand oc_eth_subenv_cfg eth;
   ...
endclass: test_cfg
```

```
...
class blk_env extends vmm_ral_env;
   test_cfg              cfg;
   oc_eth_subenv         eth;
   ral_block_oc_ethernet ral_model;
   ...
   wb_ram                ram;
   wm_mam                dma_mam;
   ...
   function new();
      super.new();eth_sub
      this.cfg = new;
      this.ral_model = new();
      this.ral.set_model(this.ral_model);
      ...
   endfunction: new
   ...
   virtual function build();
      super.build();
      ...
      this.eth = new("Eth", this.cfg.eth, this.end_vote,
                     blk_top.mii.passive, null,
                     this.ral_model, this.ram, this.dma_mam);
   endfunction: build
   ...
endclass: blk_env
```

Step 3: Configuration

The sub-environment configuration descriptor must contain all information necessary to configure the elements in the sub-environment.

SubEnv/eth_subenv.sv

```
class oc_eth_subenv_cfg;
   rand mii_cfg    mii;
   rand bit [47:0] dut_addr;
   rand bit [ 7:0] n_tx_bd;
   rand bit [15:0] max_frame_len;
   rand int unsigned run_for_n_frames;
   ...
endclass: oc_eth_subenv_cfg
```

Structural configuration is implemented in the constructor. It can be determined by the value of the virtual interfaces or channels. For example, the optional reference to a channel controls whether or not a generator is instantiated inside the sub-environment. This was implemented as part of Step 2. Structural configuration also includes appropriately configuring the transactors instantiated during construction.

SubEnv/eth_subenv.sv

```
class oc_eth_subenv_cfg;
   rand mii_cfg      mii;
   rand bit [47:0]   dut_addr;
   rand bit [ 7:0]   n_tx_bd;
   rand bit [15:0]   max_frame_len;
   rand int unsigned run_for_n_frames;
   ...
endclass: oc_eth_subenv_cfg

class oc_eth_subenv extends vmm_subenv;
   ...
   eth_frame_atomic_gen src;
   mii_monitor          mon;
   frmwr_emulator       frmwr;

   function new(string                 inst,
                oc_eth_env_cfg         cfg,
                vmm_consensus          end_test,
                virtual mii_if.passive mii_tx,
                eth_frame_channel      tx_chan,
                ral_block_oc_ethernet  ral,
                wb_ram                 dma_ram,
                vmm_mam                dma_mam);
      super.new("OcEth SubEnv", inst, end_test);
      ...
      if (tx_chan == null) begin
         this.src = new("OcEth SubEnv Src", inst);
         tx_chan = this.src.out_chan;

         this.src.stop_after_n_insts = this.cfg.run_for_n_frames;
         this.src.randomized_obj.src = this.cfg.dut_addr;
         this.src.randomized_obj.src.rand_mode(0);
       end
      ...
      this.mon = new(inst, 0, this.cfg.mii, this.mii_sigs);
      this.frmwr = new(inst, 0, this.cfg, tx_chan, this.sb,
                       this.ral, this.dma_ram, this.dma_mam);
    endfunction: new
   ...
endclass: oc_eth_subenv
```

Functional configuration is implemented in the user-defined vmm_subenv::configure() method. This step configures the DUT to match the configuration of the sub-environment.

SubEnv/eth_subenv.sv

```
class oc_eth_subenv extends vmm_subenv;
   ...
   task configure();
```

```
      vmm_rw::status_e status;

      ral.HUGEN.set(0);
      ral.FULLD.set(1);
      ...
      ral.TXEN.write(status, 1);
      if (status !== vmm_rw::IS_OK) begin
         `vmm_error(ral.log, ...);
      end

      super.configured();
   endtask: configure
   ...
endclass: oc_eth_subenv
```

The call to *super.configured()* at the end of the configure() implements an error detection mechanism to ensure that a sub-environment is completely configured before it is started.

The configure() method of the sub-environment is then invoked from at the cfg_dut() step.

SubEnv/blk_env.sv

```
class blk_env extends vmm_ral_env;
   ...
   virtual task cfg_dut();
      super.cfg_dut();
      this.eth.configure();
   endtask: cfg_dut
   ...
endclass: blk_env
```

The configuration of the Memory Allocation Manager instance that is responsible for managing the system memory must match the configuration of that memory. This can be implemented by constraining the configuration of one to match the configuration of the other.

SubEnv/blk_env.sv

```
class test_cfg;
   ...
   rand wb_slave_cfg  ram;
   rand vmm_mam_cfg   mam;

   constraint test_cfg_valid {
      ram.port_size   == wb_cfg::DWORD;
      ram.min_addr    == 32'h0000_0000;
      ram.max_addr    == 32'hFFFF_FFFF;
       ...
      mam.n_bytes      == 4;
```

```
      mam.start_offset == ram.min_addr;
      mam.end_offset   == ram.max_addr;
   }
   ...
endclass: test_cfg
```

Step 4: Execution Sequence

The sub-environment must then be integrated in the block-level environment's execution sequence. First, the vmm_subenv::start() method must be extended to start all transactors and ancillary threads in the sub-environment.

SubEnv/eth_subenv.sv

```
class oc_eth_subenv extends vmm_subenv;
   ...
   virtual task start();
      super.start();

      if (this.cfg.run_for_n_frames > 0 && this.src != null) begin
         this.src.start_xactor();
      end
      this.frmwr.start_xactor();

      fork
         forever begin
            eth_frame fr;
            this.mon.to_phy_chan.get(fr);
            this.sb.received_by_phy_side(fr);
         end
      join_none
      ...
   endtask: start
   ...
endclass: oc_eth_subenv
```

The start() method of the sub-environment must then be called in the extension of the vmm_env::start() method.

SubEnv/blk_env.sv

```
class blk_env extends vmm_ral_env;
   ...
   virtual task start();
      super.start();
      ...
      this.eth.start();
      ...
   endtask: start
```

```
   ...
endclass: blk_env
```

The *vmm_subenv::stop()* and *vmm_subenv::cleanup()* methods must be similarly implemented then invoked from the *vmm_env::stop()* and *vmm_env::cleanup()* method extensions respectively.

SubEnv/eth_subenv.sv

```
class oc_eth_subenv extends vmm_subenv;
   ...
   virtual task stop();
      super.stop();

      if (this.src != null) this.src.stop_xactor();
   endtask: stop

   virtual task cleanup();
      super.cleanup();

      if (this.cfg.run_for_n_frames > 0) begin
         `vmm_error(this.log,
                    $psprintf("%0d frames were not seen by the
scoreboard",
                              this.cfg.run_for_n_frames));
      end
   endtask: cleanup
endclass: oc_eth_subenv
```

SubEnv/blk_env.sv

```
class blk_env extends vmm_ral_env;
   ...
   virtual task stop();
      super.stop();
      this.eth.stop();
   endtask: stop

   virtual task cleanup();
      super.cleanup();
      this.eth.cleanup();
   endtask: cleanup
endclass: blk_env
```

The most important—and difficult to implement—is the wait_for_end step. Fortunately, the VMm Environment Composition package includes a new object that helps make the decision whether to end the test or not: vmm_consensus. An instance of that object already exists in the *vmm_env::end_vote* class property and it has already been passed into the *vmm_subenv::end_test* property via the constructor of the sub-environment (See Step 2).

The decision to end the test is now centralized in that one object but the contributors to that decision can be distributed over the entire verification environment. The sum of all contributions will be used to determine whether to end the test or not, regardless of how many contributors there are.

From the sub-environment's perspective, the test may end once the generator is done, all channels are empty and all transactors are idle. There is no wait-for-end method in the *subenv* base class as the end-of-test decision is implemented through the *vmm_consensus* instance. The various contributions to the end-of-test consensus are registered in the extension of the *vmm_subenv::start()* method.

SubEnv/eth_subenv.sv

```
class oc_eth_subenv extends vmm_subenv;
   ...
   virtual task start();
      super.start();

      if (this.cfg.run_for_n_frames > 0 && this.src != null) begin
         this.start_xactor();
      end
      this.frmwr.start_xactor();

      fork
         forever begin
            eth_frame fr;
            this.mon.to_phy_chan.get(fr);
            this.sb.received_by_phy_side(fr);
         end
      join_none

      if (this.src != null) begin
         this.end_test.register_notification(this.src.notify,
                                             eth_frame_atomic_gen::DONE);
      end
      this.end_test.register_channel(this.tx_chan);
      this.end_test.register_channel(this.mon.to_phy_chan);
      this.end_test.register_xactor(this.frmwr);
      this.end_test.register_xactor(this.mon);
   endtask: start
   ...
endclass: oc_eth_subenv
```

From the block-level environment's perspective, the test may end when the sub-environment consents and all transactors and channels instantiated in the block-level environment itself are respectively idle and empty. The consent of the sub-environment is automatically taken care of by virtue of using the same *vmm_consensus* instance. The other contributions to the end-of-test decision are registered in the extension of the *vmm_env::start()* method.

SubEnv/blk_env.sv

```
class blk_env extends vmm_ral_env;
   ...
   virtual task start();
      super.start();
      ...
      this.eth.start();

      this.end_vote.register_channel(this.host.exec_chan);
      this.end_vote.register_xactor(this.phy);
      this.end_vote.register_xactor(this.host);
   endtask: start
   ...
endclass: blk_env
```

The implementation of the *wait_for_end* step is now only a matter of waiting for the end-of-test consensus to be reached.

SubEnv/blk_env.sv

```
class blk_env extends vmm_ral_env;
   ...
   virtual task wait_for_end();
      super.wait_for_end();
      this.end_vote.wait_for_consensus();
   endtask: wait_for_end
   ...
endclass: blk_env
```

Step 5: Block-Level Tests

The block level environment now completely integrates the sub-environment and is ready to perform block-level tests. For example, a trivial block-level test would limit the number of transmitted frames to perform initial debug.

SubEnv/blk_trivial_test.sv

```
program test;
   blk_env env = new;

initial
begin
   env.gen_cfg();
   env.cfg.eth.run_for_n_frames = 3;
   env.run();
end
endprogram: test
```

The test may be run using the following command:

Command:

```
% make blk_tx
```

Step 6: System-Level Environment

Constructing the system-level environment using two instances of the sub-environment is very similar to constructing the block-level environment.

First, two configuration descriptors are required instead of one.

SubEnv/sys_env.sv

```
class test_cfg;
   rand oc_eth_subenv_cfg eth[2];
   ...
endclass: test_cfg
```

Two instances of the sub-environments are required, each connected to their respective physical interfaces. A single RAL model, RAM and Memory Allocation Manager is shared across both sub-environment because there are shared resources.

SubEnv/sys_env.sv

```
class sys_env extends vmm_ral_env;
   ...
   oc_eth_subenv eth[2];
   ...
   virtual function build();
      ...
      this.eth[0] = new("Eth0", this.cfg.eth[0], this.end_vote,
                        sys_top.mii_0.passive, null,
                        this.ral_model, this.ram, this.dma_mam);

      this.eth[1] = new("Eth0", this.cfg.eth[1], this.end_vote,
                        sys_top.mii_1.passive, this.src.out_chan,
                        this.ral_model, this.ram, this.dma_mam);
      ...
   endfunction: build
   ...
endclass: sys_env
```

Notice how the end-of-test *vmm_consensus* object is passed to both sub-environments. This allows both sub-environments to independently contribute to the end-of-test decision.

Also, notice how the *tx_chan* value is not *null* for the second instance of the sub-environment. This configures the sub-environment to use the external source provided—in this case an externally instantiated atomic generator.

Both sub-environments must be configured. It is a good idea to fork the configuration steps to perform as much of it as possible concurrenly.

SubEnv/sys_env.sv

```
class sys_env extends vmm_ral_env;
...
   virtual task cfg_dut();
      super.cfg_dut();
      fork
         this.eth[0].configure();
         this.eth[1].configure();
      join
   endtask: cfg_dut
   ...
endclass: sys_env
```

The *start()*, *stop()* and *cleanup()* methods must similarly invoke their respective counterpart in both sub-environment instances.

SubEnv/sys_env.sv

```
class sys_env extends vmm_ral_env;
   ...
   virtual task stop();
      super.stop();
      this.eth[0].stop();
      this.eth[1].stop();
   endtask: stop
   ...
endclass: sys_env
```

However, the *wait_for_end()* task remains identical as the *vmm_consensus* object takes care of scaling the end-of-test decision, regardless of the number of contributors.

SubEnv/sys_env.sv

```
class sys_env extends vmm_ral_env;
   ...
   virtual task wait_for_end();
      super.wait_for_end();
      this.end_vote.wait_for_consensus();
   endtask: wait_for_end
   ...
endclass: sys_env
```

Step 7: System-Level Tests

The system-level environment now completely integrates two sub-environments and is ready to perform system-level tests. For example, a trivial system-level test would limit the number of transmitted frames to perform initial debug.

SubEnv/sys_trivial_test.sv

```
program test;
   sys_env env = new;

initial
begin
   env.gen_cfg();
   env.cfg.eth[0].run_for_n_frames = 3;
   env.cfg.eth[1].run_for_n_frames = 3;
   env.run();
end
endprogram: test
```

The test may be run using the following command:

Command:

```
% make sys_tx
```

Step 8: Congratulations

You have now completed the development and integration of a reusable sub-environment. You can now speed-up the development of system-level tests by leveraging the work done in block-level environments.

You may consider reading other publications in this series to learn how to write VMM-compliant command-layer transactors, verification environments, integrate a Register Abstraction Layer model or a Data Stream Scoreboard, or use the Memory Allocation Manager.