Getting off the ground when creating an RVM test-bench

Rich Musacchio, Ning Guo

Paradigm Works

rich.musacchio@paradigm-works.com,ning.guo@paradigm-works.com

ABSTRACT

RVM compliant environments provide reusability, modularity and extensibility. However, the RVM specification does not cover every detail for developing such an environment and can be somewhat overwhelming when read initially. This paper addresses a couple of topics that are not specified in much detail and are typically needed early on in test-bench construction. The intent of the paper is to aid new adopters in getting off to a smooth start in their test-bench construction.

Table of Contents

Table of Figures

1 Introduction

The RVM specification is a comprehensive document that describes how to construct a constrained random type of test-bench that employs the use of a coverage driven methodology and promotes the re-use of code. This paper is meant to aid those who are implementing the RVM and would like to have a starting point for creating a test-bench based on this specification. Two functional areas will be discussed which include how to create a straight forward compile and run process, and how to model and access Control and Status Registers (CSR).

2 Creating a straight forward compile and run process

The RVM specification does not specify any particular process for compiling and running simulations. The specification contains some examples using Makefiles for compiling and running simulations but nothing that could be considered using on a larger scale. It is desirable to create a methodology that is easy to maintain and presents a uniform 'look and feel' for compiling and running simulations. Maintaining consistency is important so that support for multiple test-benches and the many options for compiling and running simulations is facilitated. To that end the following top level goals are presented:

- Use of a common directory structure
- Script based user interface for compiling and running simulations
- Support for Vera and NTB

The sections included in chapter 2 will discuss what can be done to achieve these goals.

2.1 Use of a common directory structure

Part of the process for compiling and running simulations is establishing a directory structure. Specifying where source files reside and where output is placed is an important part of this process. Team members are exposed to the directory structure every time they use the test-bench. The directory structure is probably one of the most 'visible' aspects of the creation of a verification environment. There are many structures that can be used and many opinions of what the best one is. The most important goal is to select a common directory structure that can be used for all test-benches based on the RVM specification. Adopting the directory structure in the RVM specification is a good starting point. One additional reason to use this structure is to facilitate the integration of third party VIP that use the RVM specified directory structure into the environment.

During the process of putting the files used in the verification environment together, one realizes that there are a number of components that do not have a 'place' in the RVM directory structure. Examples of these type files would include code that is shared among VIPs and projects. There are at least three types of code that needs to be commonly accessible in order to minimize redundancy and make the environment easier to maintain. The following sections will discuss what these components were and suggestions for where to place them.

2.1.1 VIP level common code

The RVM directory structure defines a verif/vips directory in which components such as a PCI transactor or other interface level transactors reside. In addition one would find third party VIP in this directory. The components found at this level of the directory structure are generally shared among projects. There is an additional category of code not specified by the RVM specification that may be shared between the VIP's components. Examples of this type of code are base packet classes, business unit base classes, intermediate transactor classes such as segmentation and reassembly engines, and lastly algorithms and functions such as CRC generators/checkers. It is

convenient to put these components into a ‘common’ directory at the same level as the verif/vips directory, i.e. verif/vips/common. Additionally, it makes sense to create sub-directories to allow sorting into logical places. Classes that are considered ‘data’, i.e. they extend rvm_data could be put into verif/vips/common/data. A packet class that is shared would be an example of this. Classes that are considered ‘transactor’, i.e. they extend rvm_xactor could be put into verif/vips/common/xact. A segmentation engine would be an example of this type of code. Lastly there may be common algorithms such as CRC generation/checking, 8b10b encoding and decoding. These algorithms can be put into classes which can then be placed in the verif/vips/common/misc directory.
Figure 1 illustrates a directory structure that can be used.

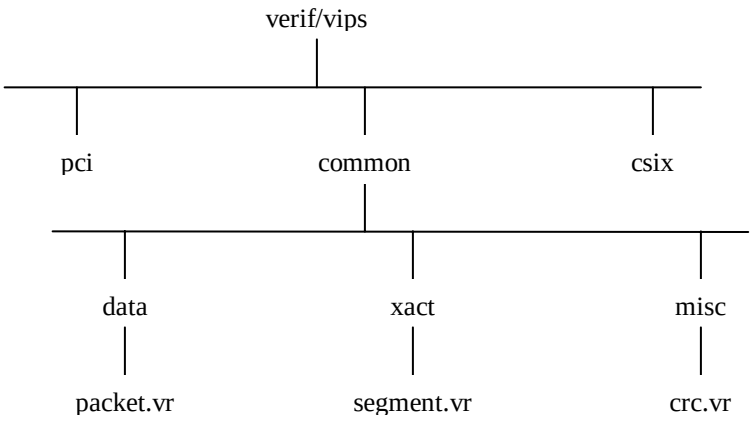

Figure 1: Directory Structure for VIP shared code

2.1.2 Project level common code

There is often project level code that is shared among multiple test-benches. Clock modules, register classes, DUT configuration descriptors, back door load facilities are examples of this type of code. This code can be either Vera or Verilog but where it resides in the directory structure is not addressed in the RVM specification. In order to facilitate access to this type of code by multiple test-benches one can create a verif/common directory. Additional sub-directories for vera and verilog can be used under the verif/common directory as well. Figure 2 illustrates a suggested directory structure.

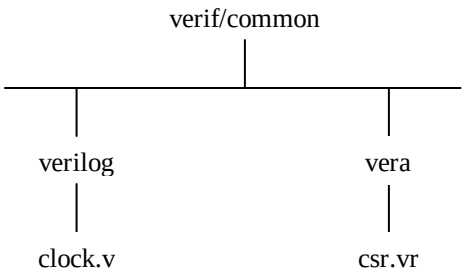

Figure 2: Directory Structure Project Level Code

2.1.3 Simulation output

There are a number of files that are created when compiling and running simulations. The types of files that are produced could include compile logs, test output logs, coverage reports, executables for the test-bench, .vro and .vrh files etc. The RVM specification does specify a verif/blk/tests/abc/logs directory but this may be insufficient for a number of reasons. The main reason is that in general the output files that are produced take up more disk space than the source files. For this reason it may be prudent to direct simulation output, i.e. test logs, wave files, coverage reports to a directory path separate from the verif/blk/tests/abc/logs directory where there is sufficient disk space to store them. We would also suggest that the directory path include root name, workspace name, test name and seed number so that a unique directory is used by each test seed as shown below.

root_name/workspace_name/test_name_seed

The RVM specification also specifies that all .vro files should go into verif/vips/vros. The main reason for reconsidering this is interoperability with NTB. When compiling with NTB there is no .vro or .vrh file produced. In addition NTB does not support a '3 step compile process' i.e. separate compile for test-bench, test and Verilog at the time this paper was written. The test-bench and test must be compiled at the same time with NTB. This means that all executables must be put into unique directories if one needs to run multiple tests simultaneously. When using NTB as the simulator all executable files (as well as other output files) can be put into the directory shown above. When using Vera one can chose to direct all .vro and .vrh files to a sim directory in verif/dut/env. This is suggested in order to keep all the .vro and .vrh files for the VIP's, test-bench, and test in one convenient directory where simulations are run from.

2.2 Script based compiling and running of simulations

The RVM specification does not require any particular process for compiling and running simulations. There are some references to using Makefiles but they appear to be there so a user can become familiar with switches used for compiling and running with Vera and NTB. Many verification environments do use Makefiles for compiling and running simulations. In cases where incremental compile is desired and the tools being used do not directly support incremental

compile, this may make sense. Makefiles sometimes have a 'reputation' for being finicky and prone to a good deal of maintenance. In addition supporting many options such as waveform viewing, debugging, etc. are more difficult to implement with Makefiles. Perl scripts are a good choice for creating a user interface for compiling and running simulations. One script can be created to compile and run all test-benches. Arguments to the script are used to define which test-bench to compile and run, what test to run, etc. The script would then form the appropriate command line arguments for Vera, VCS or NTB, compile and run the simulation, and direct output as noted in section 2.1.3.

2.2.1 Taking advantage of built in incremental compile of tools

When using Vera, VCS or NTB one can take advantage of the incremental compile features built into the tools. VCS and NTB have supported incremental compile for Verilog for quite some time and in recent times support has been added in Vera. In addition the RVM specification states in chapter 1: "All source files implementing an environment should be compiled using the -dep_check and –F options". In order to support this requirement the compile/run script should create a list of Vera compile options that include the –dep_check switch when compiling the Vera source files. For the Verilog portion of the compile the script can build up a list of Verilog compile options of which –Mupdate can be used. With the use of these switches one should be able to support full incremental compile without the use of Makefiles.

2.2.2 Defining what to compile

One of the main functions of the compilation process is to define which files to compile and where to find the files. The RVM specification defines where to find files in the directory structure definition and it also gives some indication of how to define what to compile. As previously mentioned, the RVM specification states in Chapter 1: "All source files implementing an environment should be compiled using the –dep_check and –F options".

Vera and NTB support compilation through a file list. The names of the source files to compile are put into a file. The name of this file is then communicated through the use of the –F switch. Note the convention used for file lists, i.e. what files are contained in each file list, is not explicitly defined. One can compile the entire test-bench by building up a hierarchical structure of file lists. The 'top' of the file list hierarchy can be put in the verif/dut/env directory and can be named dut_env.list. This file contains the name of all file lists corresponding to the test-bench components used by this test-bench. It can also contain source file names to compile that are unique to this test-bench. Each of the 'major' test-bench components, i.e. those found in the verif/vips directory that correspond to a transactor, can contain a file list containing the source files needed to compile that particular VIP. This facilitates putting test-benches together that contain different components. If a Perl script is used to compile the test-bench it would add –F dut_env.list to the Vera or NTB compile command line.

One advantage of using this methodology is the minimization of compile order dependencies. A number of us have run into environments where problems were encountered with compile order dependencies. Through the use of the file lists these type of dependencies can be minimized.

2.2.3 Three step compile process

One of the most powerful aspects of a constrained random test-bench is the ability to run a particular test with many seeds in order to cover as many corner cases as possible with the same base test. In order to take advantage of this capability an environment needs to be able to compile and run multiple tests and seeds simultaneously from the same workspace. A three step compilation process in the compile/run script can facilitate this. The three compile steps that were followed are:

1. Compile the test-bench: This step compiles all test-bench related source files and also includes creating a dummy vhsell.
2. Compile the test: This step compiles the test and creates the 'actual' vshell since it includes the 'real' program block.
3. Compile the Verilog: This step compiles all the Verilog.

After the three step compile is completed, the compile/run script can 'run the test' by executing the simv with all run-time arguments. In order to do this the script should create a list of all object files to use in a file called the env.vrl file. This file is part of the RVM specification. The script can then run the simulation using the +vera_mload=env.vrl runtime argument. This is an area where we believe a script based approach is more straightforward as compared to the RVM specification which states that "There shall be a Makefile target to simulate each individual test case".

2.3 Support for Vera and NTB, heading toward System Verilog

In the long term a team may want to employ a process that would facilitate using System Verilog in the future. NTB can be used as an intermediate step toward this goal. The follow sections are included to highlight some of the things that are needed to support compiling and running with Vera and NTB.

2.3.1 Declaration of externally referenced classes

We suggest following the guidelines in the document "Vera to NTB guidelines" concerning the declaration of externally referenced classes. This includes a #include of a header file for each externally referenced class. These header files can be automatically generated by Vera. However in NTB there is no automatic header generation so externally referenced classes need to be declared as extern. Each class should use a #ifdef to determine whether to use the header file or extern declaration. Figure 3 shows an example of this:

```
#ifdef SYNOPSYS_NTB
extern class contraption;
extern thingamabob;
#else
#include  “contraption.vrh”
#include “thingamabob.vrh”
#endif


class widget {

contraption      blue_contraption;
thingamabob    red_thingamabob;

}
```

Figure 3: Referencing external classes

2.3.2 Two step compile process for NTB

In section 2.2.3 we discussed a three step compile process that can be used to facilitate the compiling and running of multiple tests/seeds concurrently from the same workspace. At the time of this paper, NTB did not support this functionality, i.e. separate compile for the test-bench and test. In order to support the goal of running multiple tests/seeds concurrently in the same workspace one needs to come up with an interim process when using NTB. The solution we would suggest is to use a two step compile process that compiles the test-bench and test into the same executable, and the Verilog into a separate executable. The ramification of this process is that the executable for the test-bench/test can no longer be shared so it has to be located in a different directory. We suggest putting this executable along with all the .vro and .vrh files into the test log directory. While inefficient in terms of disk space consumption, it allows one to compile/run multiple tests/seeds concurrently from the same workspace.

2.3.3 Script changes for NTB support

In order to support both Vera and NTB in the compile/run script, an argument can be added to the script to select which simulator to use. The script can default to one simulator and be overruled with the use of this argument when the other simulator is desired.

There is a fairly big difference in the syntax used for specifying compile and runtime arguments with Vera and NTB. These differences can be incorporated into the script. Most of the syntax differences are in the use of –ntb_opts in the switch name when specifying the compile options to

NTB. In addition the –I switch used by Vera to specify the search path for #include files changed to –ntb_incdir for NTB.

3 Modeling and accessing CSRs

CSR (control and status register) accesses are required for initializing a DUT, verifying interrupts, checking statistics, and so on. The RVM specification does not present any guidelines for how to model the CSRs. In addition it is not clear from the specification how one represents this information in a configuration descriptor class and how to pass this information to the DUT.

3.1 Modeling of CSRs

CSRs are used to configure the DUT to a proper operational mode, and to track status updates. In order to verify a DUT, the test-bench needs to program the DUT to a particular mode by accessing the CSRs. Sometimes the test-bench also needs to check the status registers within the DUT. All these processes require the verification environment to provide the appropriate access methods for reading and writing the DUT CSRs.

3.1.1 CSR access methodologies

There are two commonly employed methods to access the DUT CSRs:

- Back door access
- Front door access through a transactor

Back door access can be done through Verilog/VHDL tasks built into the DUT or through direct access of the wires which are internally connected to the DUT CSRs. The advantage of back door access is that it consumes less simulation time when programming the DUT. Some of the drawbacks include dependence on the RTL designers to provide back door access support, and no coverage of the register interface.

Front door access is done through a transactor which issues read/write requests to the DUT. One advantage of the front door access is that it can be used to configure the DUT, and to test aspects of the DUT register interface at the same time. The drawback is that a long amount of simulation time can be consumed when programming the DUT CSRs during initialization.

3.1.2 CSR structure

Regardless of the approach used for accessing CSRs, the verification environment has to pass values to and from the CSRs. A CSR is identified by its distinctive address, associated with a fixed length bit vector (i.e., the value) which could be further divided into bit slices to represent a particular configuration/status setup. By accessing CSRs, the verification environment will be able to set a particular value to a specific CSR address, or get the value from a specific CSR address. Usually, each CSR is also assigned a name so one can easily identify the CSR.

3.1.3 Modeling of CSR

In order to pass a value to or from a CSR, one needs to provide the address and data for each CSR access. Both the address and the data are bit vectors. This can be very error prone if one has to type in all these bit vectors by hand for every CSR access. In addition it is also difficult to tell what configuration the DUT is programmed to by looking at a bit vector. Considering the CSR structure, we see that every CSR consists of address and data, but data is divided into a different number of slices for each CSR. In addition, the size of each slice and the attribute of each slice, such as, read-only, read-to-clear, etc., are different for every individual CSR. This type of structure can be efficiently represented by a Vera class object. All the information we just described for a CSR structure can be modeled by class member fields in the object. Methods for accessing the members of the object would allow the external environment to modify or inquire about details of the CSR more conveniently. For example, a CSR model class can define methods such as *set_field (name)* or *get_field (name)* to pass an individual register field value.

3.1.4 Creating CSR model classes

Since there are a number of common features shared by CSRs, it would make sense to create a base class to represent the common features and access methods. Below is a brief description of some features that are convenient to have in a base CSR model class in addition to the register address and data fields:

- Setting/resetting register value
- Modifying individual register field
- Packing field values to register value
- Unpacking register value to individual field values
- Shadowing actual status of register
- Modeling attributes such as read-clear, write-one-clear, etc

The DUT CSRs are extended from the base class to inherit the common features and methods and to customize other features and methods. A derived class is created for each of the DUT CSRs.

The number of CSRs in a DUT can be quite large especially for a complex system. Creating all the model classes effectively is a challenge. There are a number of ways to do this. One method is script based. This method uses a script, perhaps using a language like Perl, to convert a CSR description from a text format or excel spreadsheet to the Vera classes for each CSR. Another method is to create a Vera macro to parse strings and build the classes. The following section describes an implementation that uses a Vera macro to read in CSR information such as register name, address, bit slices (i.e. fields), and create the CSR classes as extensions of the base register class. An example of a Vera macro call used to create a register class for the register named control_1 is shown below.

csr_macro(*control_1, adddress, “field1”, “field2”, “31:24”, “23:0”* **)**;

Each register in the DUT that needs to be accessed would have a corresponding Vera macro call that is used to create the actual class. A top level file is used as a placeholder for all the Vera macro calls for the DUT CSRs. Lastly a class object constructs all the DUT CSRs and the methods used to initialize the DUT. This class is referred to as the 'DUT register map'. Any transactor or object that needs to access the DUT CSRs would be passed a handle to the register map class object in its constructor. Figure 4 shows an example of the pieces used.

```
#ifdef SYNOPSYS_NTB
extern class base_csr
#else
#include base_csr.vrh
#endif

csr_macro(reg1,…)
csr_macro(reg2,…)
csr_macro(regn,…)
```

Top level placeholder

```
class dut_register_map {

reg1   reg1_inst;
reg2   reg2_inst;
regn   regn_inst;

cfg_desc cfg;

 task new (string instance, cfg_desc cfg) {
   // Whatever the constructor needs to do

   // Handle to configuration descriptor
   this.cfg = cfg;
 }

 task init_sequence_1 () {
   reg1_inst = new;
   reg1_inst.set_field("field1", cfg.mode);
 }
}
```

DUT Register Map

Figure 4: CSR support files, class

3.2 DUT Initialization

In order to initialize a DUT properly, the verification environment needs to generate a valid set of configuration parameters and pass the parameters to CSRs. The configuration parameters are usually represented in a higher level of abstraction than the actual CSR representation. The configuration parameters are usually part of the environment configuration descriptor or the DUT

configuration descriptor. In fact the RVM specification states “The configuration descriptor shall model the configuration, not the register values”. In some implementations the configuration parameters are part of the DUT configuration descriptor and are instanced in the environment configuration descriptor. These parameters need to be generated, i.e. randomized before the initialization routines for the DUT are called. In the RVM specification the environment configuration descriptor is randomized in the rvm_env::gen_cfg() task.

Once the environment and DUT configuration parameters have been randomized the CSRs can be updated to reflect the values that are to be programmed into the DUT. The RVM specification states “The generated value of the configuration descriptor shall be downloaded into the DUT in the rvm_env::cfg_dut_t() method”. Figure 5 shows an example of the code used to randomize the configuration descriptor and the subsequent loading of the DUT CSRs. In order to meet the RVM requirement the ‘load_csr_values” task in the DUT register map is called when entering the rvm_env::cfg_dut_t() task. This task ‘downloads’ all the values of the DUT configuration descriptor to the DUT CSRs. The next operation in the rvm_env::cfg_dut_t() task passes off the appropriate commands to program the DUT using back door or front door accesses. Note that there is a bit in the environment configuration descriptor that is used to indicate front door versus back door access and initialization. This bit is constrained in the env_cfg class to whichever mode is desired. As stated previously for most simulations the back door access may be preferable due to the saved simulation time. However some subset of tests must be run using the front door access in order to verify the DUT CSR interface.

There was an interesting detail that one needs to consider. The RVM specification calls for all transactors to be started in the rvm_env::start_t() task and for the DUT to be initialized within the rvm_env::cfg_dut_t() task. However if one chose to use a front door access the transactor used for programming the DUT is not started until after the rvm_env::cfg_dut_t() task is called. So when using front door access a choice needs to be made between starting the transactor used for front door access within the rvm_env::cfg_dut_t() task , or the rvm_env::start_t() task followed by the actual programming of the DUT. Either choice is valid.

Another consideration is whether to wait for the initialization of the DUT to complete before starting up any other transaction generation. The choice to wait for DUT initialization to complete before allowing other transaction generation is dependent on whether the DUT needs to be verified in this mode or not.

```
  virtual task gen_cfg() {
    super.gen_cfg();

    rvm_note(log, "Randomizing configuration descriptor");
    void = this.cfg.randomize();
}

virtual task cfg_dut_t() {
    super.cfg_dut_t();

    // Load CSR values from configuration descriptor
    rvm_note(this.log, "Loading Randomized CSR Values into CSR class");
    dut_reg_map.load_csr_values();

    // Test back door bit in cfg to determine if we should do back door or not
    if (this.cfg.backdoor_load == 1) {
      // Back door load enabled
      rvm_note(this.log,"Back door loading is enabled....");

      foreach (dut_reg_map.dut_regs,i) {
          rdnwr = 0; // Always do a write
          address = dut_reg_map.dut_regs[i].reg_addr;
          wr_data = dut_reg_map.dut_regs[i].get_reg_data();
          // Call the back door load task
          dut_backdoor_load(rdnwr,address,wr_data,rd_data,fail);
    }
    else {
      // Queue up DUT CSR accesses to transactor that does front door load
       xactor.queue_transactions(dut_reg_map.dut_regs);
    }
}
```

Figure 5: Example code for CSR loading

4 References

Reference Verification Methodology User Guide, version 8.5.0, May 2004

5 Glossary

This section lists the terms, and acronyms used in this paper.

CSR: Control and Status Register
DUT: Device Under Test
NTB: Native Test Bench
RTL: Register Transfer Language
RVM: Reference Verification Methodology
VIP: Verification Intellectual Property