Command Line Debug Using UVM Sequences

Mark Peryer
Mentor Graphics (UK) Ltd
Rivergate, London Rd., Newbury,
Berkshire, U.K. RG14 2QB
Telephone: +44 1635 811614
mark_peryer@mentor.com

ABSTRACT

The mainstream use case for the UVM is to create a verification environment that supports the running of multiple test cases which run sequence based stimulus and use automatic checking and coverage mechanisms to achieve closure on a verification plan. However, there is another important use case which is not so well addressed and that is the interactive debug of hardware and test bench bugs. This paper describes a technique which leverages UVM sequences to implement a command line debugger which can be used to facilitate efficient hardware debug, and potentially, the debug of the sequences themselves. In practical terms, the technique is encapsulated in a command line debug sequence package.

Using the package allows a user to debug designs more efficiently by being able to interactively run short, targeted tests and to check the result using the debug resources of the simulator. The approach is well suited to the early stages of hardware debug or integration debug where its interactivity allows users to track down simple, but blocking issues, very quickly. The technique is lightweight, leverages existing UVM sequences and is orthogonal to the main UVM use model of building up regressions of test cases.

Categories and Subject Descriptors

[Hardware Verification]: Functional Simulation and Verification – *Class based SystemVerilog, UVM class library, sequences, register model, command line debug.*

General Terms

Verification, Simulation, Sequences.

Keywords

UVM, Sequences, sequence wrappers, testbench architecture, debug techniques, Registers

1. INTRODUCTION

The Universal Verification Methodology (UVM) is a SystemVerilog base class library which allows users to construct verification environments using verification component objects and to create stimulus using sequence objects. The UVM encapsulates a number of standard building blocks with well-defined APIs which makes it straight-forward for users to collaborate or to build testbenches which are interoperable with third party verification IP.

1.1 The UVM Verification Process

The UVM makes a clear separation between stimulus generation and the structure of the verification environment. The structure of the UVM test bench is created during the build phase when the various components are configured and constructed, these structural components stay in place for the lifetime of the simulation. However, stimulus generation is by means of a library of sequence classes which have a transitory life time, being created, executed and de-referenced as required. Each UVM simulation starts by constructing a test class that is responsible for defining the configuration of the test bench structure, starting the top-down build process and then starting the execution of the chosen set of sequences.

The typical UVM test bench architecture is orientated around the interfaces of the Design under test (DUT), with each hardware interface having a verification component, or agent, associated with it. The agent contains a driver which is responsible for converting abstract transaction objects called sequence_items into signal level activity. Stimulus is generated for an interface inside sequence objects which create and shape a series of sequence_item objects and send them to the driver via component object called a sequencer. Co-ordinated stimulus between hardware interfaces is achieved through the use of supervisory sequences which execute sequences on multiple sequencers, these are usually referred to as virtual sequences and they are executed via virtual sequencer component objects which contain handles for the target interface sequencers. This architecture is illustrated in Figure 1.

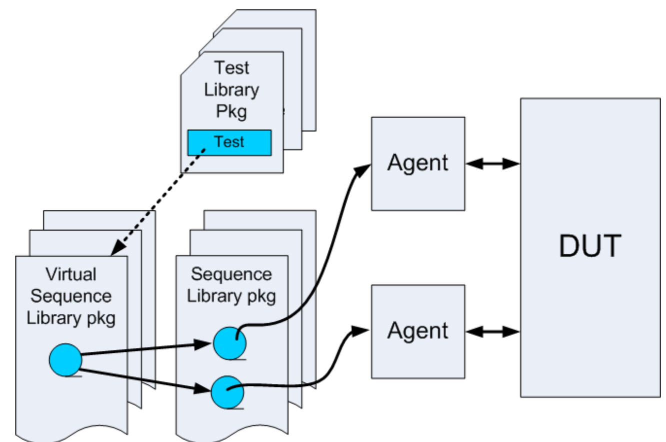

Figure 1 – UVM Test Architecture

Once the UVM verification architecture has been established, the verification process centres around the creation of test cases based on the use of sequences. The emphasis of the verification process is on generating volumes of stimulus as efficiently as possible using constrained random techniques; re-running test cases with different seeds to create conditions which flush out corner case bugs in the DUT. When aligned with a metric based, coverage driven verification process the UVM has proved to be an effective means of verifying hardware using simulation techniques.

Once a test case has passed its initial debug phase, it is passed into a regression suite which is run in batch mode. In the regression suite the test case is run with different seeds to catch corner cases. The priority is on simulation throughput to explore as many parts of the design state space in a given time. This use model is efficient once

the test bench architecture has been completed and once the DUT has reached a reasonable level of maturity, but there is a considerable period of time where a more interactive use model can facilitate rapid debug. This is where the use of sequences controlled from the command line can prove effective.

1.2 Command line debug sequence use model

A simulator converts a hardware description of a DUT into an executable model. In the classic UVM use model, the emphasis is on wrapping the DUT with a test bench and generating stimulus that effectively runs in batch mode with minimal interaction from the user, unless a bug in either the DUT or the verification environment is uncovered. The command line use model takes advantage of the UVM test bench infrastructure and its stimulus objects, but places the user in the driving seat by allowing him to interactively direct what stimulus is applied to which interface of the executable DUT model, and in what order. The closest parallel to this is bringing up real hardware on a lab bench with a software debug monitor, signal generators and instruments. The architecture of the command line debug environment augments the typical UVM environment as illustrated in Figure 2.

In the early stages of bringing up a DUT, there are often simple problems which need to be debugged interactively with the verification engineer and the designer working through a series of directed steps to understand what is wrong with the hardware. The complexity of these steps will range from individual reads and writes of hardware registers to running a series of complex stimulus patterns.

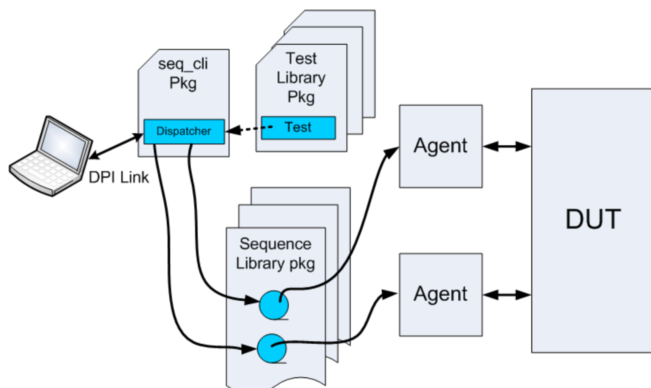

Figure 2 – Command Line Debug Using UVM Sequences

A typical scenario would be to run an initialisation routine, then check the state of the DUT registers, followed by a DUT operation, followed by a dump of the DUT register or memory content. Obviously, this can also be done via the mainstream UVM use model, but using the command line allows the user to try out different register programming options without having to rewrite, recompile and reload the UVM sequence code. With larger scale verification environments it can take a significant amount of time to iterate round an experimental sequence loop and it can be tricky to debug, the command line approach allows the user to try an experiment several times until the solution is identified without having to change any underlying code or reload the simulation.

2. IMPLEMENTATION

The implementation of the command line debug infrastructure leverages the structure of the UVM verification environment and the sequence library packages developed for the DUT. The main infrastructure of the system is generic, but there are parts of the implementation which need to be customised to address the specifics of an environment. A script has been developed to reduce the extraction and customization effort.

2.1 Requirements

Before implementing the command line debug system the following requirements were considered:

2.1.1 Ease of use

The debug system needed to be straight forward to use, the target being to provide a hardware designer with no knowledge of UVM a useful debug environment.

2.1.2 Low implementation overhead

The time taken to create an implementation of the debug system needs to be short, and the creation process needs to be agile to cope with updates to the UVM verification code.

2.1.3 Capability to run sequences

The command line interface needs to be able to specify and run sequences from the available library.

2.1.4 Capability to pass arguments

There needs to be a means of passing arguments from the command line which configure the sequences to be run. For instance, a sequence which does a bus write will require an address and a write data argument.

2.1.5 User help system

The user needs to be able to view information on the available sequences, their purpose and the command line required to execute them.

2.1.6 Capability to run command files

The command line system needs to be able to read text files which contain lists of sequence commands. These files can be used to save the user from having to enter commonly used sequences of commands at the command line.

2.1.7 Log file generation

Recording the commands submitted via the command line interface and their result allows the session to be recorded and to be used as a reference or as the basis of a command file. This facility also needs to have the capability to annotate comments.

2.1.8 Capability to run parallel commands

There are situations where there is a need to run multiple sequences as parallel threads. An example would be a sequence that sends serial packets to the DUT on one interface running concurrently with a sequence that handles the packets on another.

2.1.9 Library of utility commands

The command line package also needs to contain commands which allow the user to specify behavior not related to sequences. These include:

- wait_for_clock(n) – Wait for n system clocks
- wait_for_interrupt – Wait for an interrupt
- exit – To exit the command line debug

The main purpose of the command line interface is to execute sequences. This means that the meta-language used does not need to have a rich feature set. Complex constructs such as decision loops

and support for specifying randomization constraints are not required since they can easily be taken care of in a custom sequence, implemented in native SystemVerilog.

2.2 Implementation detail

The command line interface package comprises a SystemVerilog package and a library of c functions. The c functions implement the command line monitor. The SystemVerilog package contains the command line interpreter and tasks which create and execute the sequences. The c code and the SystemVerilog code communicate via the SystemVerilog Direct Programming Interface or DPI.

2.2.1 Using the SystemVerilog DPI

The DPI provides a light-weight way of allowing SystemVerilog code to execute c functions in an external shared object and for c functions in an external object to call SystemVerilog tasks or functions. In the SystemVerilog code, SystemVerilog tasks and functions are exported, and the c functions are imported. The SystemVerilog compilation process generates a c header file which has to be included in the c code to allow the SystemVerilog tasks and functions to be referenced; otherwise the c source code does not require any special coding techniques. The interface allows c and SystemVerilog data types to be passed as arguments.

The simulator side of the DPI needs to be implemented in the static part of the SystemVerilog language so that the shared object can be loaded and linked with the simulator kernel at the start of the simulation. This means that any DPI code has either to be in a static SystemVerilog interface or module, or it needs to be in a package. In this case, a sequence command line package was written and this encapsulated the DPI imports and exports as well as the tasks and functions required to implement the interface. The c side of the DPI was implemented as a series of c function calls.

Any sequence of DPI calls has to start under the control of the SystemVerilog code. The command line debugger is started by the OVM test making a call through the DPI to a c function called SV_debugger() which is the command line monitor loop. The simulation is then blocked until the user makes an entry at the command line, at which point the string entered is passed to the SystemVerilog code to interpret and dispatch the selected sequence wrapper task with the appropriate arguments. Once the command has been executed on the simulation side, control is passed back to the c code to pick up the next command to be executed. The prompt loop is terminated when the user types 'exit', at which point the SV_debugger() function returns and the OVM test run method can continue.

2.2.2 The SystemVerilog package (cli_seq_pkg.sv)

The SystemVerilog side of the debug package is contained within a package. The package imports the UVM package (uvm_pkg); the packages containing the sequences which it references; and the packages for the target agents on which the sequences will be run. Inside the package are a small number of generic tasks and functions associated with processing the command line:

- process_cmd_line() – Function that is called by the command dispatcher, it parses a command line string and returns a data structure
- process_cmd_file() – Task to open and read a command file and pass each line to the command dispatcher class
- dispatcher() – Task which takes the command line and dispatches the relevant sequence(s) for execution
- virtual sequencer handle

The standard DPI calls, i.e., export or import statements, are also present in the package.

The rest of the package is specific to an UVM verification environment and is generated by a script based on the content of the target sequence packages. It comprises the following tasks and functions:

- Sequence wrapper classes – one per target sequence
- Init() – Task that constructs all the sequence wrapper objects, assigns sequencer handles and puts them into an associative array.
- virtual sequencer – Containing handles for the target sequencers that the sequences will run on.
- Utility functions – specific to an environment

2.2.3 The sequence wrapper class

The sequences used by the command line debugger need to be created, configured and then started from the cli_seq package. In order to use a generic command dispatcher, the sequences are encapsulated in a wrapper class, derived from a base class, which contains methods to simplify the implementation of the package. Sequences are wrapped and then placed in an associative array, indexed by a string. The wrapper classes are generated by a script which parses the sequence packages.

To illustrate how a wrapper class is implemented consider a bus read sequence:

```
class apb_read_seq extends ovm_sequence #(apb_seq_item);
`ovm_object_utils(apb_read_seq)
//-------------------------------------------
// Data Members (Outputs rand, inputs non-rand)
//-------------------------------------------
rand logic [31:0] addr;
logic [31:0] data;

// Standard OVM Methods:
extern function new(string name = "apb_read_seq");
extern task body;
endclass:apb_read_seq

function apb_read_seq::new(string name = "apb_read_seq");
  super.new(name);
endfunction

task apb_read_seq::body;
  apb_seq_item req = apb_seq_item::type_id::create("req");;

  begin
    start_item(req);
    req.we = 0;
    req.addr = addr;
    finish_item(req);
    data = req.data;
  end

endtask:body
```

This sequence has been coded following the recommended coding style where configuration data members are randomizable (addr) and that result data members are not (data). This coding convention is assumed by the generation script.

The code for the sequence wrapper base class is as follows:

```
// Sequence wrapper class:
//
// Takes a sequence and wraps it with other helper methods
//
virtual class sequence_wrapper extends uvm_object;

function new(string name = "sequence_wrapper");
  super.new(name);
endfunction
```

```
// Handle for the target sequencer
uvm_sequencer_base sqr;

// Template for the help messaging:
virtual function string help();
  `uvm_error("sequence_wrapper",
             "Help method not implemented")
endfunction: help

// Returns the number of arguments used by run_sequence
virtual function int no_args();
  return 0;
endfunction: no_args

// Needed to spawn multiple sequences concurrently
virtual function sequence_wrapper spawn();
  sequence_wrapper c = new();

  c.sqr = this.sqr;
  return c;
endfunction: spawn

// Assign the handle for the target sequencer
virtual function void set_sqr(
                             ovm_sequencer_base sequencer);
  sqr = sequencer;
endfunction: set_sqr

// Task to start the sequence from the wrapper
// Generated code goes here ...
virtual task run_sequence(int arg0 = 0,
                          int arg1 = 0,
                          int arg2 = 0,
                          int arg3 = 0);
  `uvm_error("sequence_wrapper",
             "run_sequence method not implemented")
endtask: run_sequence

endclass: sequence_wrapper
```

The resultant wrapper class that is generated for the example apb_read_sequence is as follows:

```
// This code is generated
class apb_read_seq_wrapper extends seq_wrapper;

`uvm_object_utils(apb_read_seq_wrapper)

function new(string name = "apb_read_seq_wrapper");
  super.new(name);
endfunction

function string help();
  return "apb_read_seq addr; - target sequencer: APB";
endfunction: help

function int no_args();
  return 1;
endfunction: no_args

function sequence_wrapper spawn();
  apb_read_seq_wrapper c;

  c = new();
  c.sqr = sqr;
  return c;
endfunction: spawn

task run_sequence(int arg0 = 0,
                  int arg1 = 0,
                  int arg2 = 0,
                  int arg3 = 0);
  apb_read_seq seq = apb_read_seq::type_id::create("seq");
  seq.addr = arg0;
  seq.start(sqr);
  $display("apb_read: addr:%0h data:%0h", arg0, seq.data);
  $fdisplay(log_fh,  "apb_read:  addr:%0h  data:%0h",  arg0,
seq.data);
endtask: run_sequence

endclass: apb_read_seq_wrapper
```

The package contains an initialization function which is responsible for constructing each of the wrapper tasks and putting them into an associative array indexed by a string which corresponds to the name of the sequence. When a command line request is received by the command line dispatcher, it looks up the wrapper sequence in the associative array, calls the spawn() method to create a new deep copied object and then executes it. The spawning is required to enable multiple copies of a sequence to be run in parallel.

The sequence does not return any data to the command line because it cannot process returned values, however the result of the sequence execution is displayed and recorded in the log file:

```
# apb_read_seq: addr: 0 data:0
# apb_write_seq: addr:0 data: aaaaaaaa
# apb_read_seq: addr:0 data:aaaaaaaa
```

2.2.3 The package generation script

In order to make the package code relatively painless for the user to produce, a generation script has been developed. The script works in two phases.

The first phase takes a list of sequence library packages, parses the sequence descriptions and generates a reference file containing a list of target sequences and arguments that could be used with those sequences. The user then takes the reference file and comments out those sequences that he wishes to omit from the command line debug package. The user can also comment out sequence arguments and specify the name of the target sequencer.

The second phase takes the edited list and generates the wrapper tasks, the help functions, the virtual sequencer and the dispatcher tasks. The generated code is written into files which are `included into the command line debug package.

Once the reference file is in place, the process can be run again to capture any new sequences developed as the environment develops. This enables a new version of the debug package to be produced very quickly. The process can be used with ordinary sequences or with virtual sequences.

3 USING THE PACKAGE

Once the package is in place, it is compiled and imported into the SystemVerilog package that contains the test case classes. Inside any test class that is going to allow the use of the package, four things need to be done in order to get things up and running:

- The package virtual sequencer needs to be declared and constructed.
- The handles for the various target sequencers in the package virtual sequencer have to be assigned.
- The virtual sequencer handle in the package has to be assigned to the virtual sequencer in the test.
- At some point in the execution of the test run method, the package sv_debug task has to be called to enable the debugger.

The following pseudo code illustrates how the debug functionality would be integrated into an UVM test:

```
// Example leaving out irrelevant code
class with_debug extends spi_test_base;

`uvm_component_utils(with_debug)

// Debug package virtual sequencer
seq_cli_v_sqr seq_cli_sqr;
```

```
// Build method building debug pkg sequencer
function void build();
  super.build();
  seq_cli_sqr =
     seq_cli_v_sqr::type_id::create("seq_cli_sqr", this);
  // Assign handle to package virtual sequencer handle:
  seq_cli_pkg::vs = seq_cli_v;
endfunction: build

// Virtual sequencer target sequencer handle assignment
function void connect();
  super.connect();
  seq_cli_v.APB = m_env.m_apb_agent.m_sequencer;
  seq_cli_v.SPI = m_env.m_spi_agent.m_sequencer;
endfunction: connect

task run;
  #100 ns; // Wait for reset to go away
  sv_dispatcher(); // Calls the debugger
  global_stop_request();
endtask: run

endclass: with_debug
```

Once the debug mode is entered, the normal operation of the UVM environment comes to a halt, since the simulation will be blocked from advancing whilst a command is being entered, and will advance in time only when a sequence is being executed. However, the debug facilities of the simulator will remain available to the user, allowing waveform traces, single stepping through code etc to still be used.

4 APPLICATIONS

Variants of the command line debug package have been used by different users to great effect at different levels of design integration.

The package has been used with block level environments to do basic register read and write mode debug. It has also been used to prototype driver code or to develop initialization routines. It has also proved extremely useful in debugging specific scenarios where multiple interfaces need to be stimulated using several sequences in parallel.

At cluster and SoC levels of integration, the technique has proven useful for unpicking interconnect issues such as incorrect address decoding and data path issues. It has also allowed firmware to be debugged before being encapsulated in sequences or processor based binaries.

5 FURTHER DEVELOPMENTS

The current implementation of the command line debug sequence package is based on features that were present in the OVM library. These features map directly over to the UVM 1.0 base functionality, but with the new features available in the UVM it should be possible to enhance the package.

The two most promising areas are UVM registers and the UVM resources database. Some upgrade work has been recently been done to the command line debug package to take advantage of the features of the UVM register model classes. The results of the investigations into the use of the resource database are not available at the time of publication.

5.1 UVM Registers

The UVM register base classes enable the package to provide a generic set of methods which allow the user to examine and change memory and register content. The UVM register model provides a task based API for register accesses, eliminating the need for the user to provide read and write sequences specific to the target bus.

The UVM register model is structured to align with the hardware structure of the DUT, therefore the command line allows the dumping of register content on a system, sub-system, block, register or register field level simply by specifying a path. When looking at register content, the user has the option of looking at the content of the register model database (referred to as the 'mirror'); performing a front door read() access using a bus agent; or by performing a back door peek() access using the simulator data-structure. This capability enhances the debug process, since it is often inconvenient to do a front door read access which disturbs the state of the DUT hardware.

The UVM register classes also allow memories, or regions of memories to be accessed either using front or back door access methods. The command line package has been upgraded with generic commands to allow users to dump memory content; to change memory locations; to load memory with the content of a file; or to fill it with random data.

Finally, the UVM register map class allows users to access registers and memories from different interfaces, something which is important when verifying the content of SoCs using complex interconnect fabrics which have several bus master ports. The capability to specify alternative access interfaces has also been added using a command line argument.

5.2 UVM Resources

Using the UVM resource data base offers the possibility of manipulating handles to the target sequencers and the sequence wrapper objects more elegantly than was possible before.

6. CONCLUSION

The command line sequence debug package has proven to be useful to OVM users. With the advent of the UVM, there are a number of potential upgrades which should enhance the ease of use of the package.

7. ACKNOWLEDGMENTS

My thanks go to my colleagues, especially Rich Edelman for making the time to dig my early experiments with the DPI out of the mire and Adam Erickson for giving me early insight into the UVM register classes.

8. REFERENCES

[1] Edelman, R., Warmke D. 2005. Using SystemVerilog Now with DPI – Proceedings DVCon 2005

[2] Edelman, R., 2008. Sequences in SystemVerilog – Proceedings DVCon 2008

[3] IEEE 1800 SystemVerilog LRM – IEEE 2009

[4] Meyer, A., 2009. Overview of Sequence Based Stimulus Generation in OVM 2.0 – Application note. Mentor Graphics Corporation

[5] OVM 2.1.1 Users Guide. Mentor Graphics and Cadence Design Systems – 2010

[6] UVM 1.0 Users Guide – Accellera, (Awaiting publication)