Want a Boost in your Regression Throughput? Simulate common setup phase only once.

Rohit K Jain, Mentor Graphics (510-354-7407, rohit_jain@mentor.com)
Shobana Sudhakar, Mentor Graphics (503-685-1889, shobana_sudhakar@mentor.com)

I. Abstract

As the complexity of SoC designs grows, it takes longer to do functional verification. Tests run way longer than they used to - a few hours to days - and there are thousands of them to run. It is not surprising to see a long-drawn out design verification cycle.

One of the common characteristics of functional verification of today's complex designs is that they go through some sort of initial setup phase, whether it is design initialization, reset state, configuration phase, or sequence initialization. This initial setup phase is typically common for all the tests, or for a particular set of tests or a set of test configurations. In addition, this setup phase itself takes a substantially large amount of cycles with respect to overall length of the simulation for a test, and these cycles must be repeated over and over again for all subsequent tests.

This paper will discuss how to write the design so that the common initial setup phase simulation is done once and then used as a foundation to run different tests later on, including the ability to change test stimulus to simulate different test behaviors. We will also discuss what type of designs (Verilog, VHDL, SystemVerilog, UVM-based, SystemC, C/C++ models, PLI/FLI/VPI etc.) will fit in this methodology and what a designer can do to make his design fit for such methodology. Also covered: the constraints that need to be followed in order for this methodology to work, the design factors that might prevent a designer or verification teams from adopting this methodology (and how to overcome them).

We will also touch briefly upon the co-simulation verification environment (simulation-emulation) requirements and discuss how the same methodology would work with either simulation or co-simulation verification.

We will use examples from real world designs to show how this methodology can be used successfully, what changes are required to unblock some of the design factors, and elaborate on the potential gain and regression throughput that is possible.

II. Introduction

Designers of complex designs (SoC, FPGA, Processor, PCIe link etc), may go through a common initial setup phase for all their tests. This setup phase could be either executing the exact same sequence of simulation steps, or programming their design to reach the same initialization or reset state. This approach is plausible for designs that have:

a) an onboard PHY component, which requires a long initialization, calibration, and bring-up time,

b) DDR4 or similar SDRAM interface, which requires an initial calibration and training sequence before actual transfer can take place,

c) many configuration registers, such as a video/graphics controller, requiring setup after reset.

Once the tests complete this common setup/initialization/reset phase, then they would diverge to run subsequent different tests/sequences/instructions from this point onwards. This common setup phase could consume anywhere from 20%-90% of the total simulation time for a test.

It is efficient to re-factor this initial setup phase so that what is common for a set of design tests is separated out from regular simulation cycle. Thus, each test need not redundantly run this common phase. Instead this phase can be performed only once for a whole set of regression tests and the design state is checkpointed. All subsequent tests could reuse this checkpointed state from an already initialized setup phase. If a system is created to be able to perform different tests from this point onwards, it would result in a huge amount of throughput gain for a large and complex design verification system by avoiding repeated reruns of the initial setup phase for the individual tests.

III. Methodology Description

Here is how the methodology works once the user has identified a set of common simulation steps for a set of regressions tests:

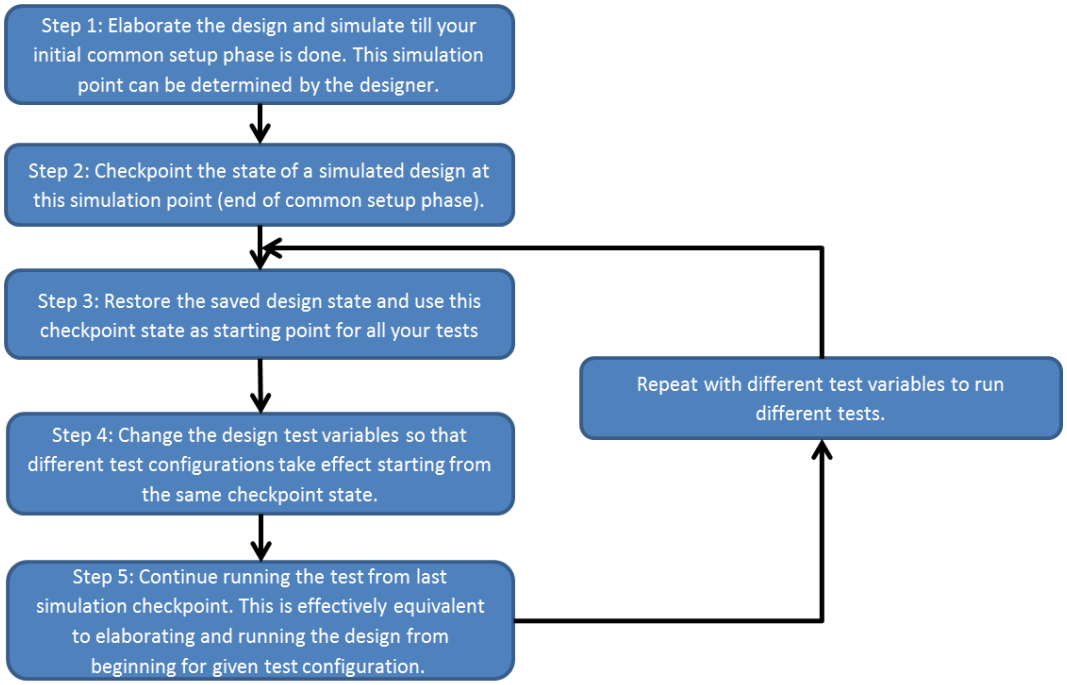

Figure 1: High-level description of steps to re-factor common initial setup phase and running different tests

This methodology can work for designs based on various types of HDLs, i.e. Verilog, System Verilog, VHDL, or a mix of them. Limited use of SystemC wrapper models (mostly used in an emulation environment) can be allowed with this methodology, also UVM/OVM-based designs.

Checkpointing and restoring a design state is proven conceptually and is supported by most of the commercial simulators [1] [2], however it has been difficult to reliably use it and maintain the flow in an extremely large and complex system that involves multiple HDL languages, C/C++ models, IPs, DPI, SystemC, PLI/FLI/VPI, etc. This paper attempts to describe how the methodology can be partially automated with the help of the simulator and requires only initial design setup and minimal modification by the designer to make it work on their existing and suitable designs.

Users can also checkpoint and restore the design state on different machines and/or on the grid machines. This allows users to integrate the methodology in their existing grid flow. However, users should ensure that the machine where design state is being checkpointed and the machine where design state is restored should have similar OS specifications. Different OS systems may have different memory mappings for low level system libraries and can cause issues during state restore. Typically grid systems, like LSF, SGE etc., have controls to allow user to submit jobs on similar types of the machines, like SLES11 or RHEL7, and prevent such issues from occurring.

This flow can be used in the co-simulation or emulation environment as well, provided the other side of the simulator connection is able to handle a checkpoint and restore of the design state in its domain. The simulator can handle the checkpoint/restore in a similar way as it does in pure simulation mode. This speeds up the setup phase on the pure simulation side so that users can get to the interesting traffic generation portion on the emulator quickly, and can get more things done in the same verification cycle.

There could still be some complications and complexities in design and/or flow, which may prompt a user to continue the verification in their existing way. But many designs can still benefit and can save simulation cycles by spending some initial time on setting up and deploying this methodology.

IV. Design modifications

If a user's design consists of only HDLs (e.g. Verilog, System Verilog, VHDL, UVM/OVM), then simulators should be able to handle Step 1-3 in the flow diagram above (**Figure 1**) without any additional intervention from the user.

If, however, a user's design contains any form of C/C++ code (shared or precompiled library, IP, C/C++ models, PLI/FLI/VPI code, DPI code), then the user or IP providers need to review the code and ensure that this code can work safely while saving the state of the design and after restoring it. Simulators can also provide a partial mechanism to automate some of these changes, so users do not have to manually do all the work. Section V below provides details on what content needs to be reviewed and changed in C/C++ code and what simulators can provide.

If a user's design contains SystemC code, then this methodology generally cannot be used. The exception (with certain restrictions) are those scenarios where SystemC wrappers over Verilog/VHDL design units are used to connect with co-simulation or emulation environments.

Step 1 & 4 of the **Figure 1** above require working with the user to figure out a good place in their design to save the simulation state, like end of initialization phase, etc., and the different variables that require changing for the different tests and how to change them. These steps may involve a few design changes. Let's discuss these steps in detail.

For Step 1, the user needs to determine the correct place in his design which can be used as the end of the common setup phase. This varies depending on the style of the design. It could be based on a fixed simulation time, or a fixed location in the testbench, or it could be dynamic and programmed based on the activity in the design.

For a fixed simulation time, a user could just run the simulation till that time, and then checkpoint the simulation state through simulator's command controls as seen in Example 1.

In Example 1, the first vsim command simulates design 'top' till '1000ns' simulation time, then checkpoints the design state in file 'test.chk' and quits the simulation. The second vsim command restores the saved design state from file 'test.chk' and continues simulation from last checkpoint simulation time.

```
//Simulate design 'top' till '1000ns', then checkpoint the state in file
//'test.chk', and quit simulation
1%> vsim -c top -do "run 1000ns; checkpoint test.chk; quit -f"

//Restore design state by loading 'test.chk' and continue simulation
2%> vsim -c -restore test.chk -do "run -a; quit -f"
```

Example 1: Command-line options on Questasim, to run simulation till 1000ns and checkpoint simulation state

For a fixed location in the testbench or dynamic activity based designs, a user could have the simulator waiting on a variable state to checkpoint the simulation state, and then just set the desired variable state inside the testbench at appropriate location, as described in **Example 2**.

In **Example 2**, the first vsim command causes testbench to set checkpoint_enable variable when the initialization phase is done. As explained in checkpoint.do file, the simulator monitors this variable and when it is set, triggers the checkpoint command execution. The simulator then saves the design state in test.chk and quits. The second vsim command restores the saved design state from file 'test.chk' and continues simulation from last checkpoint simulation time.

```
test.sv:
   task run_test();
       //... Execute initialization phase ...
       //Checkpoint design by setting checkpoint_enable variable to 1
       checkpoint_enable = 1;
       //Allow simulator to stop simulation and checkpoint the design
       #1;
       // After restore, simulation will continue from here
   endtask //run_test

# This do file is specified at simulator's command line
checkpoint.do:
    onbreak resume
    set do_checkpoint 0
    # When design sets the 'checkpoint_enable' variable to 1, trigger checkpoint command by
#setting do_checkpoint
    when {checkpoint_enable == 1} {
        echo "Stopping simulation to create a checkpoint at time $now"
        set do_checkpoint 1
        stop
    }
    while{1} {
        run -all
       # If checkpoint is triggered, save design state and quit simulation
        if{$do_checkpoint} {
            set do_checkpoint 0
            checkpoint test.chk
            quit -f
        }

1%> vsim -c top -do checkpoint.do
2%> vsim -c -restore test.chk -do "run -a; quit -f"
```

Example 2: Saving simulation state based on the state of a variable, in dynamic activity based designs

For Step 4, the user needs to determine how the test configurations change for the different tests. It could be picking up various different sequence generators, or reading test stimulus vectors from test-specific config files or something similar. Command-line plusargs syntax can be used to pass different values for different tests (**Example 3**), or change the desired test config value of specific variable, and have design code sensitive to this variable to pick up different test settings, like stimulus file, sequence class, etc. (**Examples 4 and 5).** Exact methods may vary based on capabilities and option controls provided by different simulators.

Example 3 shows the scenario where different tests configure TESTNAME and pick up different test classes based on the value of TESTNAME. The first vsim command simulates till 1000ns, checkpoints the design state in test.chk, and then continues simulation for 'test_1' test. The second vsim command restores the design and changes +TESTNAME plusargs value. This will cause simulation to continue for 'test_2' from checkpoint state with a different TESTNAME value.

```
string name;
initial begin
    #1000
    // Checkpoint here
    // After restore, Run test based on value of TESTNAME plusargs
    if ($value$plusargs("TESTNAME=%s", name))
        run_test(name);
    else begin
        uvm_report_info(get_full_name(), "No test name specified. Running default test
mytest");
        run_test("mytest");
    end
    $finish();
end

1%> vsim -c top -do "run 1000ns; checkpoint test.chk; run -all; quit -f" +TESTNAME=test_1
2%> vsim -c -restore test.chk -do "run -all; quit -f" +TESTNAME=test_2
3%> vsim -c -restore test.chk -do "run -all; quit -f" +TESTNAME=test_n
```

Example 3: Restoring simulation state with command-line plusargs in Questasim

Example 4 shows the scenario where different tests configure different sequence generators to use. The first vsim command checkpoints the design when checkpoint_enable is set to 1. The checkpoint.do file is the same as used in **Example 2** above. The second vsim command will restore the design and then change the seq_name value, which will cause the testbench to pick up a different sequence generator for subsequent tests.

```
string seq_name = "test_1"
my_test_init_seq init_seq;
my_base_test_seq test_seq;
initial begin
    init_seq = my_test_init_seq::type_id::create("initialization");
    init_seq.start(sqr);
    checkpoint_enable = 1; //Set variable to trigger checkpoint command
    #1;
    uvm_report_info(get_full_name(), $sformatf("Restored simulation with test_seq = %s",
                    seq_name));
    factory.set_type_override_by_name("my_base_test_seq",seq_name);
    test_seq = my_base_test_seq::type_id::create(seq_name);
    test_seq.start(sqr);
end

1%> vsim -c top -do checkpoint.do
2%> vsim -c -restore test.chk -do "change /top/seq_name test_2; run -a; quit -f"
```

Example 4: Restoring simulation state and modifying test config value to run different tests

Example 5 shows the scenario where different tests configure different stimulus vector files to read in. The vsim command will restore the design state and set the STIM_NAME plusargs value to test_2. This will cause simulation to pick up different stimulus test vectors for the different tests after the restore.

```
// Checkpoint here
if($value$plusargs("STIM_NAME=%s", filename))
    $display("Stimulus file name has been set to %s", filename);

// Pick test-specific stimulus file
$display($time, "Loading stimulus from file %s :", filename);
begin : read_test_file
    int fd, eof;
    string line;
    fd = $fopen(filename, "r");
    eof = !$fgets(line,fd);
    while(!eof) begin : get_line
        $display("Reading line %s", line);
        eof = !$fgets(line,fd);
    end
    $fclose(fd)
End

1%> vsim -c -restore test.chk +STIM_NAME=test_2 -do "run -a; quit -f"
```

Example 5: Restoring simulation state and modifying test stimulus file name to run different tests

Please refer to Appendix A for complete **examples 7 & 8** that show how to use the checkpoint/restore methodology.

V. C/C++ models and IP Modifications

Typically, commercial simulators are capable of saving and restoring the state of the HDL designs. However, if there is external C/C++ code (PLI/FLI/VPI/DPI/VIPs) used in the design, then users need to ensure that it follows the guidelines below for it to work with the checkpoint-restore methodology:

- All run-time memory allocation/de-allocation on the heap should be made through APIs that are provided by the simulator. For example, QuestaSim provides *mti_Malloc()* and *mti_Free()* calls that behave identically to malloc() and free() C functions, except that the memory allocated and de-allocated through the mti_Malloc() and mti_Free() calls are visible and can be managed by the simulator.

- All static or global variables as well as variables that retain their state throughout the simulation require users to checkpoint and restore them manually. Simulators may provide API routines for easy and efficient checkpoint/restore of such variables. For example, QuestaSim provides API routines to explicitly checkpoint and restore these variables and the complete set of routines is available in QuestaSim User's Manual for reference.

A C model or IP may require some code changes to enable correct checkpoint/restore methodology. All calls to malloc() and/or free() will have to be replaced with calls that do memory allocation/de-allocation that can be managed by the simulator. You can refer to the memory allocation APIs provided by the simulator and replace them in your C models.

A C++ model will require the same changes as a C model. However, some simulators may be able to handle the heap memory allocation/deallocation automatically and not require the users to manually change their code.

In addition, it may be required to explicitly checkpoint and restore any global or static variables that retain their state during simulation. **Example 6** below shows a variable 'my_scope'. This variable stores the scope of the HDL instance with the DPI calls and is intended to preserve its state (value) during the simulation. Therefore, it is required to checkpoint and restore this variable correctly.

The API routines provided by simulators can be used to checkpoint and restore these variables manually, once the variables that need to be checkpointed are identified. **Example 6** explains how these changes can be made with QuestaSim simulator.

In the **Example 6**, variable 'my_scope' can be checkpointed with a call to *mti_SaveBlock()* which is used to save a block of memory. Upon successful restore, the values can be copied into the same variable through a call to *mti_RestoreBlock().* A static constructor is executed when the C++ model is instantiated in the program, which registers the callback routine *mti_AddDPISaveRestoreCB()* to save and register this variable.

```
#include "mti.h"
//Begin existing user code
svScope my_scope;  // This variable should preserve its value upon restore

void set_my_scope() {
    my_scope = svGetScope();  //Function called at init phase to store the scope.
}
//End existing user code

//Begin code additions for checkpoint/restore
void mySave() {  // Save my_scope during checkpoint
    mti_SaveBlock((char*) &my_scope, sizeof(svScope));
}

void myRestore() {  //Restore my_scope after restore
    mti_RestoreBlock((char*)&my_scope);
}

//Static constructor registers callbacks for checkpoint and restore
class CheckpointHandlerBootStrapper {
public:
  CheckpointHandlerBootStrapper() {
    mti_AddDPISaveRestoreCB(
      reinterpret_cast<mtiVoidFuncPtrT>(mySave),
      const_cast<char*>("myRestore"));
  }
};
static CheckpointHandlerBootStrapper checkpointhandlerbootstrapper;
//End code additions for checkpoint/restore
```

Example 6: Saving and restoring global variables using API methods available in Questasim

VI. Real-world data

This methodology has been deployed successfully at large design houses. Below is data from one of the design usages.

The design is an SoC with a PHY component. This is a UVM-based design with C/C++ IP models. This design has about 1000 tests in one regression suite. Each test takes about 10 hours to run. Time spent to simulate initial setup phase is about 2 hours for each test. Hence, the total serial time to run the regressions is ~10,000 hours. However, the regression runs on grid system, and taking 20 parallel machines in consideration, the ideal regression throughput would be ~500 hours.

We worked with the design team to make changes as described in **Example 2-6** above. The design team was then able to reduce the initial setup time of ~2 hours from all their tests and was able to achieve the regression throughput with 20 grid machines to be at ~402 hours, a saving of ~20% in their regressions.

Again, proportionally higher regression throughput improvement is possible for those designs where initial setup phase consumes a larger percentage of total simulation time.

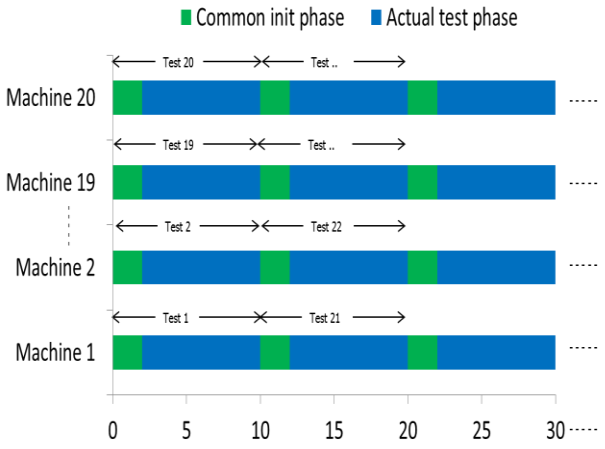

Figure 2: Regression flow of a large SoC, before re-factoring common setup phase of tests

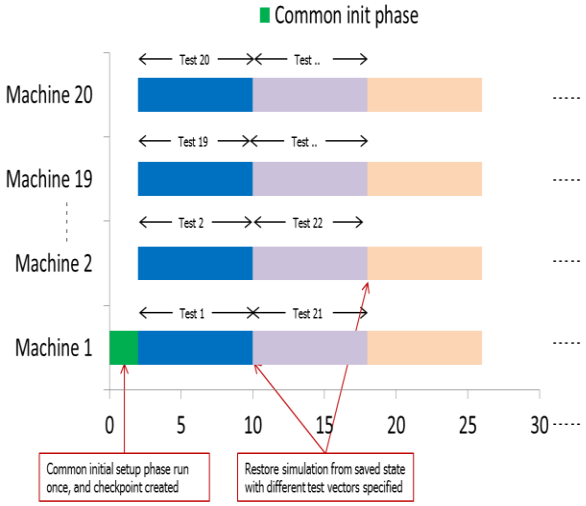

Figure 3: Regression flow after separating the common setup phase that is reused by all tests, resulting in 20% saving on regression time

VII. Conclusion

We looked at different types of designs (Verilog, VHDL, System Verilog) and use case scenarios (long running setup phases), that are suitable for this checkpoint and restore methodology. We also discussed the type of design factors (HDLs, IPs) that are suitable for this methodology and which aren't (SystemC, External C/C++). Using the simulator's automation for the methodology, the designer should be able to make some minor modifications in his design (testbench) and easily integrate this methodology in his existing design verification environment.

We also discussed the type of changes required in the design, especially in external C/C++ code and provided guidelines for designer's reference. Also, simulators can provide more automated support to handle things internally and require users to make only minimal changes at their end.

We also touched upon the fact that a similar methodology can be deployed in co-simulation or emulation environments as well.

We also discussed design data from real design scenario and the throughput that is achieved using this methodology. With design scenarios where initial setup phase time has a higher percentage of overall time, the amount of throughput speedup will be higher.

Although examples and tests output in this paper are described using QuestaSim simulator, it should be possible to achieve similar methodology with other industry simulators as well.

References

[1] "Can You Even Debug a 200M+ Gate Design?", H.Chan, B.Vandegriend, D.Joshi, C.Goss, DVCon 2013

[2] "Configuring Your Resources the UVM Way!", P.Goel, A.Sharma, R.Hasija, DVCon 2012

[3] QuestaSim 10.4 User's Manual

Appendix A - Flow Examples

Test 1:

```
package my_test_pkg;
string seq_name = "mytest";

class my_base_test_seq extends uvm_sequence#(uvm_sequence_item);
  `uvm_object_utils(my_base_test_seq);
  function new(string name="my_base_test_seq"); super.new(name); endfunction
endclass

class my_test_init_seq extends my_base_test_seq;
  `uvm_object_utils(my_test_init_seq);
  function new(string name="my_test_init_seq"); super.new(name); endfunction
  task body();
    uvm_report_info("DEBUG","running init_seq"); repeat(3000000) #1;
    uvm_report_info("DEBUG","- powering up ..."); repeat(3000000) #1;
    uvm_report_info("DEBUG","- system registers init ..."); repeat(3000000) #1;
    uvm_report_info("DEBUG","- PHY calibration ..."); repeat(3000000) #1;
    uvm_report_info("DEBUG","- DDR training ..."); repeat(3000000) #1;
  endtask
endclass

class A_seq extends my_base_seq;
  `uvm_object_utils(A_seq);
  function new(string name="A_seq"); super.new(name); endfunction
  task body(); uvm_report_info(get_full_name(),"running A_seq"); repeat(100000) #1; endtask
endclass

class B_seq extends my_base_seq;
  `uvm_object_utils(B_seq);
  function new(string name="B_seq"); super.new(name); endfunction
  task body(); uvm_report_info(get_full_name(),"running B_seq"); repeat(100000) #1; endtask
endclass

class mytest extends uvm_test;
  `uvm_component_utils(mytest);
  function new(string name="mytest", uvm_component parent=null);
    super.new(name,parent); endfunction
  uvm_sequencer#(my_base_test_seq) sqr;
  function void build(); sqr = new("sqr",this); endfunction
  task run_phase(uvm_phase phase);
    my_test_init_seq init_seq;
    my_base_test_seq test_seq;
    phase.raise_objection(this);
    uvm_report_info("DEBUG","checkpoint save/restore example");
    init_seq = my_init_seq::type_id::create("init");
    init_seq.start(sqr);
    uvm_report_info("DEBUG","init done - stopping for checkpoint - specify seq_name upon restore");
    $stop;
    uvm_report_info("DEBUG",$sformatf("restored after checkpoint with seq_name=\"%s\"",seq_name));
    factory.set_type_override_by_name("my_base_test_seq",seq_name);
    test_seq = my_base_seq::type_id::create(seq_name);
    test_seq.start(sqr);
    phase.drop_objection(this);
  endtask
endclass
endpackage
```

Example 7: Example code that shows checkpoint/restore methodology

Test 1 output with QuestaSim Simulator:

```
vsim -c test -do "run -all; checkpoint test.chk; quit -f"

        # UVM_INFO @ 0: reporter [RNTST] Running test mytest...
        # UVM_INFO @ 0: uvm_test_top [DEBUG] checkpoint save/restore example
        # UVM_INFO @ 0: uvm_test_top.sqr@@init [DEBUG] running init_seq
        # UVM_INFO @ 3000000: uvm_test_top.sqr@@init [DEBUG] - powering up ...
        # UVM_INFO @ 6000000: uvm_test_top.sqr@@init [DEBUG] - system registers init ...
        # UVM_INFO @ 9000000: uvm_test_top.sqr@@init [DEBUG] - PHY calibration ...
        # UVM_INFO @ 12000000: uvm_test_top.sqr@@init [DEBUG] - DDR training ...
        # UVM_INFO @ 15000000: uvm_test_top [DEBUG] init done - stopping for checkpoint - specify
        seq_name upon restore
        # ** Note: $stop    : test1.sv(51)

vsim -c -restore test.chk -do "change /my_test_pkg::seq_name test2; run -all;
quit -f"

        # Simulation kernel restore completed
        # Restoring graphical user interface: definitions of virtuals; contents of list and wave
        windows
        # env sim:/uvm_pkg::uvm_task_phase::execute/#FORK#137(#ublk#215181159#138)_7feff0a2404
        # Stopped at test1.sv line 51
        # sim:/uvm_pkg::uvm_task_phase::execute/#FORK#137(#ublk#215181159#138)_7feff0a2404
        # change /my_test_pkg::g_seqname test2
        # run -all
        # UVM_INFO @ 15000000: uvm_test_top [DEBUG] restored after checkpoint with seq_name="test2"
```

Test 2:

```
  initial begin
    $display("Running Checkpoint Save/Restore test - testname is \"%s\"",testname);
    $display("Running Initialization Sequence:");
    do_reset();    do_count_up(16);
    $display("Stopping at end of initialization for checkpoint:");
    checkpoint_enable = 1;
    #1;
    if ($value$plusargs("TESTNAME=%s", testname))
        $display("Restored after checkpoint: testname has been set to \"%s\"",testname);
    else $display("testname is not set");
    $display($time, " Loading test file \"%s\":",testname);
    begin :read_test_file
      int fd, eof; string line;
      fd = $fopen(testname,"r");
      eof = !$fgets(line,fd);
      while (!eof) begin :getline
        $display("- read line \"%s\"",line);
        eof = !$fgets(line,fd); end
      $fclose(fd);
    end

    do_count_up(4);
    $display("End of main test"); $display("Finishing now:");
    $finish;
  end
```

checkpoint.do:

```
onbreak resume
set start_checkpoint 0
when {checkpoint_enable == 1} {
        echo "stopping simulation to do checkpoint at $now"
        set start_checkpoint 1
        stop
}
while {1} {
        run -all
        if {$start_checkpoint} {
                set start_checkpoint 0
                checkpoint test.chk
                quit -f
        }
```

Example 8: Example code that shows checkpoint/restore methodology

Test 2 output with Questasim Simulator:

```
vsim -c test -do "checkpoint.do"

        # Running Checkpoint Save/Restore test - testname is "do-nothing"
        # Running Initialization Sequence:
        # Stopping at end of initialization for checkpoint:
        # stopping simulation to do checkpoint at 170
        # Simulation stop requested.

vsim -c -restore test.chk +TESTNAME=test2 -do "run -a; quit -f"
        # Simulation kernel restore completed
        # Start time: 23:38:05 on Dec 15,2014
        # Restoring graphical user interface: definitions of virtuals; contents of list
        and wave windows
        # env sim:/test
        # sim:/test
        # run -a
        # Restored after checkpoint: testname has been set to "test2"
        # 171 Loading test file "test2":
```