Improving Verification Productivity of Embedded C Tests Using the VCS Save/Restore Feature

Neel Sonara (nsonara@broadcom.com)
Amir Nilipour (amirn@synopsys.com)
Ajay Thiriveedhi (akrishna@synopsys.com)

ABSTRACT

Verification engineers still use simulation as the primary mechanism to validate the functionality and quality of designs. Even with advanced verification methodologies, faster compiled simulators, and expansive compute farms, the design and verification teams still encounter long simulation and regression turnaround times, which can adversely affect project schedules. To reduce the expensive engineering time spent in developing and running a large suite of native tests, the use of simulator technologies that reduce total simulation turnaround time is critical. This paper discusses the bottlenecks to turnaround time when using Embedded C testing for full-chip SoC verification. Using the Save and Restore feature in VCS, a synchronization scheme is adopted between the C side and the simulator to identify common saving points to be reused for different C tests. All aspects of the deployment of this feature, including integration into mainstream testing as well as the multiplicative time savings achieved, are presented.

Table of Contents

1 Introduction 3

2 Verification Environment for Embedded Tests 3

2.1 WHERE DO I USE EMBEDDED C TESTING? 3

2.2 DETAILS OF EMBEDDED C TESTING ENVIRONMENT 3

3 Verification Challenges 4

3.1 TEST DEVELOPMENT AND DEBUG TURNAROUND TIME 4

3.2 REGRESSION TURNAROUND TIME 5

4 VCS Save/Restore 5

4.1 SAVE/RESTORE USE MODELS 5

4.1.1 Save/Restore from Simulator Command Line Interface (UCLI) 5

4.1.2 Save/Restore from Testbench 6

4.1.3 Reseeding with Save/Restore 6

4.2 SAVE/RESTORE APPLICATIONS FOR EMBEDDED C TESTING 7

4.2.1 Develop and Debug New C Tests Faster 7

4.2.2 Achieve Faster Regression Turnaround Time 8

4.2.3 Combine Static Save/Restore and Dynamic Interactive Checkpoint/Rewind 9

4.2.4 Multiple Save Points 10

5 VCS Save/Restore Feature Implementation on an Embedded C Testing Environment 10

5.1 HOW NATIVE C CODE IS PARTITIONED AND WHERE IT RESIDES 11

5.2 HOW SYNCHRONIZATION BETWEEN C AND SIMULATOR IS ACHIEVED FOR SAVE POINT .. 12

5.3 HOW RESTORE IS IMPLEMENTED 13

5.4 BEST PRACTICES 14

6 Conclusions 14

7 References 15

Table of Figures

Figure 1 Compile Flow of C Test and Scatter File 4

Figure 2 Example of Scatter File with Multiple Functions 4

Figure 3 Flexible FSDB Schemes Used With Save/Restore 8

Figure 4 Verdi's Interactive Checkpoint and Rewind Feature 10

Figure 5 Identifying Common Execution Points Amongst C Tests 11

Figure 6 Processor's Memory Map and Location of Scatter File 12

Figure 7 Identify the Event to Notify Vcs to Create Snapshot ..... 13~~12~~

Figure 8 Restore a Saved Snapshot, Load New Test and Run ..... 14~~13~~

1 Introduction

Verification teams often encounter long debug and regression turnaround times and constantly look for opportunities to gain an upper hand over these challenges. If such opportunities can be discovered during the planning phase, prior to the project launch, the entire team will reap the benefits of an expedited project schedule and improved design quality at a reduced overall cost of simulation licenses and CPU farms. Once identified, it often requires some front-end work to develop and integrate into the mainstream verification flow. However, the benefits outweigh the initial efforts throughout the verification of the design for RTL, power-aware and gate level netlist simulations.

We found that the Synopsys VCS Save/Restore feature is an ideal aspect to be considered, due to the maturity of this basic feature and its proven capability.

In an ideal scenario, there should be no simulation time wasted on redundant testing, even if it constitutes only a portion of individual tests. Since the bulk of our SoC testing is via Embedded C tests, the flow would have to understand the structure of such tests and the common pieces of C code for which their re-execution would need to be avoided at all cost. The degree to which this can be achieved relies on how the flow is implemented. For example, it can allow partitioning of a test to create multiple intermediate saved points and also detecting when an appropriate saved image can be restored. These topics are discussed in this paper.

We also describe the methodology guidelines and implementation details of a synchronization scheme between the C side and the simulator, as they are not part of the standard Save/Restore. Even though it is specific in nature, we strongly believe many ideas can be borrowed if users share similar embedded verification environments.

2 Verification Environment for Embedded Tests

2.1 Where Do I use Embedded C Testing?

The SoC architecture for today’s mobile devices consists of multicore application processors, audio subsystem, multimedia, modem subsystem, power manager, on-chip memory, additional peripherals and interfaces to standard buses, and possibly off-chip memory. Some type of on-chip bus, bus fabric, or network-on-chip connects all the units together. Since the embedded processors run software, the complete SoC is the chip plus the code that runs on these processors. Many of the full chip scenarios interact with different subsystems, interrupts, power up/down require simulation with a processor running the C code. These tests are developed and reused at gate level and power-aware simulations.

2.2 Details of Embedded C Testing Environment

The Embedded C test environment includes files (set up files and library files containing useful functions) that are compiled initially. As illustrated in Figure 1, the C test is compiled and is linked using a scatter file to create memory images before simulation is run. The scatter file has a memory map of the processor and provides flexibility to keep different C code in different memory sections. The memory images created are preloaded at the beginning of the simulation and the processor starts fetching and executing after reset is removed.

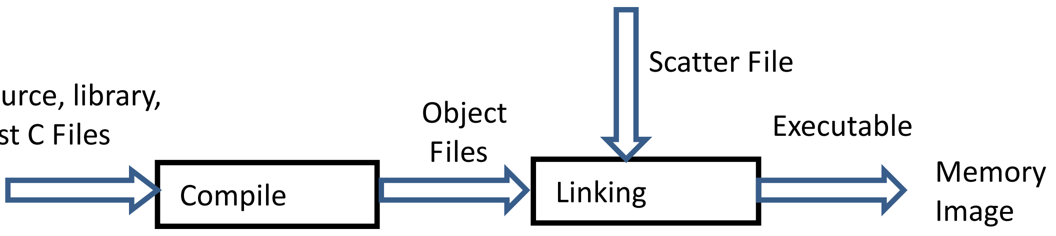


Figure 1 Compile Flow of C Test and Scatter File

Figure 2 shows the implementation of a sample scatter file that contains multiple functions.

```
SRAM_LOAD 0x34060000 NOCOMPRESS 0x8000 {
    MY_SECTION 0x34060000 0x7000 {
        *      (MASTER_SECTION)
        .ANY (+RO, +RW)
    }
    XYZ_EXEC 0x34067000  0x1000 {
        .*    (XYZ_AREA)
        .ANY (+RO, +RW)
    }
}

DDR_LOAD 0x80000000 NOCOMPRESS 0x00F00000 {
    DDR_DATA 0x80000000 ZEROPAD 0x00F00000 {
        *    (DDR_AREA)
    }
}
```

Figure 2 Example of Scatter File with Multiple Functions

In a typical Embedded C-based verification environment, the start-up code initializes the stack pointer, cache, MMU, clock gating, DDR, etc. This code is common for most of the tests and after this initialization, the program flow branches to the main test.

3 Verification Challenges

3.1 Test Development and Debug Turnaround Time

A long SoC initialization sequence takes simulation time in milliseconds and runtime in 20-40 minutes before the actual test starts. Verification engineers typically need to rerun a test multiple times during test development before it is added to the regression. This iteration is required as the engineer adds more checks to make the tests robust, and he/she may need to add more program code that was missed before, for example, enabling clock gating. Lots of time is wasted in the initialization sequence before meaningful actual test code starts to execute. Turnaround time for the test is a key factor in the overall ability of being able to develop multiple tests in a short span of time.

3.2 Regression Turnaround Time

To verify the robustness of bug fixes and new functionalities, a full regression suite has to be run. Due to long initialization and setup time for every test, a meaningful regression run takes a long time, especially for gate level and power-aware simulations. On average, execution of an initialization sequence can take more than an hour.

4 VCS Save/Restore

Synopsys VCS provides users with the capability to save a simulation snapshot at a specific time. This saved snapshot could then be used in several different simulations to replay the simulation until that point. This feature is commonly referred to as the Save/Restore feature in VCS and helps users reduce the total simulation time for several verification test cases that share a common start-up sequence. A typical example of this start-up sequence is the reset sequence, which is shared by all the test cases. Third party applications such as Specman and Denali memory models already have Save/Restore functionality built-in to work with VCS.

4.1 Save/Restore Use Models

The VCS Save/Restore feature could be used in several ways. Options available are to write conditional code in the testbench to save the simulation snapshot or to save and restore it from the VCS simulator command line interface, known as UCLI. Users typically save and restore from the UCLI when debugging a specific test. The following sections show how to control Save/Restore from either UCLI or testbench.

4.1.1 Save/Restore from Simulator Command Line Interface (UCLI)

The following steps demonstrate how to use Save/Restore from simulator command line interface.

STEP1: Compile the design.

```
      vcs  <compile_options> <design_testbench_files>
```

STEP2: Run the simulation from the UCLI, and save the snapshot at a specific time. Note that the extension of the saved snapshot file does not matter.

```
       simv –ucli
       UCLI% run 100us
       UCLI% save common_sequence.chk
       UCLI% run 300us
```

STEP3: Restore the saved sequence when you rerun the test. Note that the design/testbench should be the same between both tests. In other words, the simulation executable simv should be the same.

```
       simv –ucli
       UCLI% restore common_sequence.chk
```

```
        UCLI% run 500us
```

4.1.2 Save/Restore from Testbench

In a regression use model or even if users don't exactly know when to perform Save/Restore, it is possible to use this feature from a testbench. The following show the steps that need to be followed.

STEP1: Implement the save functionality in the testbench. The following shows a way to do this using $test$plusargs approach.

```
       if ($test$plusargs("save")) begin
         $save("common_sequence.chk");
        end
```

STEP2: Compile the design.

```
       vcs  <compile_options> <design_testbench_files>
```

STEP3: Run the simulation to save the snapshot.

```
      simv +save
```

STEP4: Restore the saved sequence in another test. Note that the design/testbench should be the same between both tests. In other words, the simulation executable "**simv" s**hould be the same.

```
      simv –r common_sequence.chk
```

4.1.3 Reseeding with Save/Restore

SystemVerilog-based testbench environments, such as VMM and UVM, typically run the same test with multiple seeds to ensure that all of the corner case scenarios are exercised for that test. The VCS Save/Restore functionality has been enhanced to reduce the total simulation time for N runs of the same test. In a conventional simulation, the seed for the simulation is set at the beginning of the simulation, and the same seed applies for the entire simulation. With this feature, VCS provides users with the capability to reseed a simulation after the saved state is restored.

The following demonstrates how to use the reseeding feature in VCS along with the Save/Restore feature using UCLI interface.

STEP1: Compile the design.

```
       vcs  <compile_options> <design_testbench_files>
```

STEP2: Run the simulation from the UCLI, and save the snapshot at a specific time.

```
  simv –ucli
        UCLI% run 100us
        UCLI% save common_sequence.chk
        UCLI% run 300us
```

STEP3: Restart the simulation with a new seed specification or with an automatic seed that can be generated by the tool.

```
    simv +ntb_random_reseed +ntb_random_seed=13
or
    simv +ntb_random_reseed +ntb_random_seed_automatic

    UCLI% restore common_sequence.chk
    UCLI% run 500us
```

4.2 Save/Restore Applications for Embedded C Testing

4.2.1 Develop and Debug New C Tests Faster

In a typical scenario in which new C testcases are being developed, the actual focused test starts after a long and already well verified SoC initialization sequence that, on its own, can take many milliseconds of simulation time. Ideally, one needs not wait for re-execution of this initialization sequence in order to continue developing a new testcase or start root causing a failed one.

In our experience, once the Save/Restore methodology was deployed, a C test developer would, on average, get a 30-minute jumpstart for each debug iteration and reach the point of failure in just a matter of minutes. Also, keep in mind that once a test fails, it often would need to be rerun with FSDB wave dump on, which can exacerbate the time it takes to complete the boot up sequence before the actual test begins. The Verdi integration to the simulator does not necessitate wave dumping to be turned on prior to the save point. The user has the option of enabling it any time after restore, around the time of failure for a partial dump, or simply right after restore. This flexibility gives us substantial savings in not only simulation runtime, but also in disk space saving, since for a golden and verified checkpoint, FSDB dump is not required. To enable this feature, add **`-ucli2Proc`** to the simulation runtime options during both the save and restore runs.

Depending on the waveform dumping scenarios of interest, one can do any of the following. For each case, assume two simulations are run serially. The first is run until a suitable save point is reached and saved. From that point, the second simulation starts and runs to the end of the test.

The colored FSDB file names in the following scenarios correspond to the ones used in Figure 3.

1. Turn FSDB on prior to save in the first simulation, and leave dumping on during the second simulation. The FSDB data will be auto-merged from both runs into the same FSDB file. The screenshot below shows the file **`all.fsdb`**.

   UCLI commands for save session:
   ```
   fsdbDumpvars {"+fsdbfile+all.fsdb"}
   stop -cond {pc==10} -cont -command {run 0; save snap; quit}
   ```

   Runtime commands for restore session:
   ```
   restore snap
   run
   ```

2. The second scenario is to split the waveform into two files so the one corresponding to the save session can be used as a golden dump file and the second can be a working dump file to be used for development or debug. Both files can be loaded simultaneously into Verdi for a seamless debug spanning both simulations.

   Runtime commands for save session:
   ```
   fsdbDumpvars {"+fsdbfile+snap.fsdb"}
   stop -cond {pc==10} -cont -command {run 0; save snap; quit}
   ```

   UCLI commands for restore session:
   ```
   restore snap
   fsdbDumpoff
   fsdbDumpvars {"+fsdbfile+rest.fsdb"}
   run
   ```

3. The last scenario is to turn FSDB on only in the second simulation after restore for a situation where the initialization sequence has been debugged and the corresponding wave is not needed. This corresponds to the dump file called **`rest.fsdb`**

   UCLI commands for save session:
   ```
   stop -cond {pc==10} -cont -command {run 0; save snap; quit}
   ```

   UCLI commands for restore session:
   ```
   restore snap
   fsdbDumpvars {"+fsdbfile+rest.fsdb"}
   run
   ```

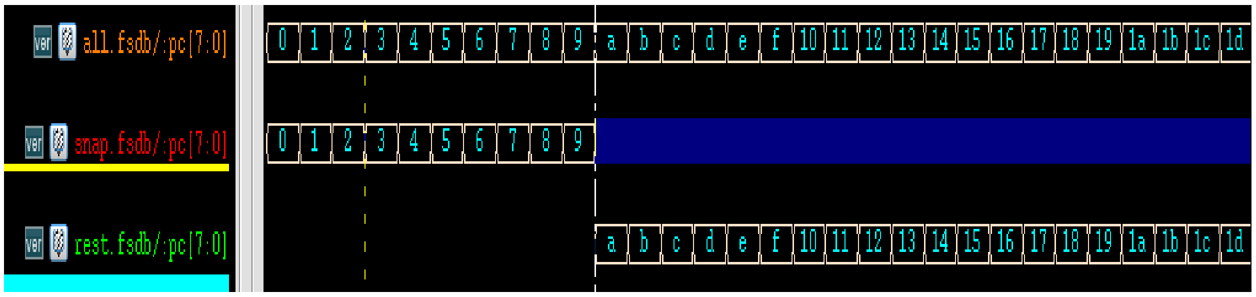

all.fsdb/:pc[7:0]
snap.fsdb/:pc[7:0]
rest.fsdb/:pc[7:0]
0 1 2 3 4 5 6 7 8 9 a b c d e f 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d
0 1 2 3 4 5 6 7 8 9
a b c d e f 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d

Figure 3 Flexible FSDB Schemes Used With Save/Restore

4.2.2 Achieve Faster Regression Turnaround Time

Each of the Embedded C tests in our large regression goes through a specific flavor of the SoC initialization for which a separate checkpoint would be required. Our initial effort was to identify, develop, and debug these various, but limited, bootup sequences and create a library of qualified snapshots for every viable sequence. As a side benefit, this process also motivated us to rethink the testing procedures to gain more flexibility and efficiency.

The regression scripting was upgraded to automatically group the tests sharing a common initialization sequence and would run each test from its corresponding saved snapshot in the library. In selecting which checkpoint to use, other simulation characteristics also had to be considered. For example, whether waves were dumped or specific SoC level monitors were enabled, etc., to match with the characteristics of the saved points.

The multiplicative regression time savings easily added up over the large number of tests, since with each, there is an almost zero boot time, except for the negligible time it takes to restore the common snapshot.

4.2.3 Combine Static Save/Restore and Dynamic Interactive Checkpoint/Rewind

The Interactive Check/Rewind feature in Verdi, allows multiple simulation checkpoints to be created dynamically. These checkpoints are not written out to files, but instead created in memory and only used during Verdi interactive debug. Subsequently simulation can be restarted (Rewind) from any of these dynamic checkpoints during the current debug session. This section covers how the static Save/Restore can be combined with this interactive feature to reduce debug cycle time even further.

Once simulation is restored to the end of the initialization sequence, run to the point of failure in the C test, and create an interactive checkpoint that can be used to reiterate through the debug cycle for faster turnaround time.

The following steps summarize the debug flow:

1. Run the standalone simulation until a suitable save point is reached, and save the static snapshot called **`snap`**. Make sure to include **`-ucli2Proc`** as a runtime option.

2. In a new simulation, bring up Verdi Interactive debug to execute commands either from the Verdi prompt or the GUI menus and buttons.

3. Restore he snapshot saved in step 1.
   ```
   SimCMD > restore snap
   ```

4. Interactively run the simulation to the point when interactive debug or what-if analysis can be used to root cause a failure.
   ```
   SimCMD > break -cond { pc == 38 }
   SimCMD > run
   ```

5. Click "**`Add Checkpoint`**" button to save the simulation image or, alternatively, use **`checkpoint`** command as:
   ```
   SimCMD > checkpoint -add myCheck
   ```

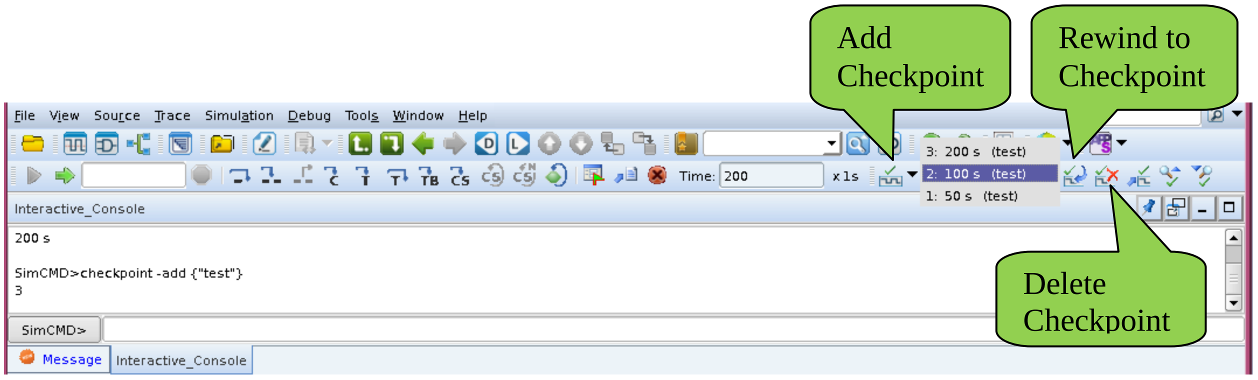
6. Continue with debug as usual while taking advantage of what-if analysis features of Verdi, such as interactive forcing (freeze/deposit) of signals or manipulating SystemVerilog constraints (update/add/delete/etc.).

7. At any point in the simulation, rewind to any saved checkpoint. To do so, use either the **`Rewind`** button or the **`checkpoint`** command.
   ```
   SimCMD > checkpoint -join -id 2
   ```

To find which **`id`** is associated with an intended checkpoint, use the **`-list`** option of the checkpoint command.

```
SimCMD > checkpoint -list
```

The following Figure 4 shows some of the Verdi menu options discussed in this section related to Checkpoint/Rewind.



```
200 s

SimCMD>checkpoint -add {"test"}
3
```

Figure 4 Verdi's Interactive Checkpoint and Rewind Feature

4.2.4 Multiple Save Points

A similar flow was used in debugging the initialization sequence itself by creating intermediate save points as we progressed towards the end of the sequence. These were only required during test development as for regressions only one checkpoint was needed at the end of each initialization sequence. This common checkpoint would subsequently be used as a restore point to launch into individual tests.

5 VCS Save/Restore Feature Implementation on an Embedded C Testing Environment

A given native C test gets compiled with all other library functions and linked to create the memory images containing all instruction opcodes corresponding to the test. These images are then loaded into ROM and RAM using Verilog's **`$readmemh`** task before reset is released, so they can be fetched and executed by the processor after reset. Many, if not all, of the tests in our regression can be categorized into groups that share an initial common native code. This common code contains an initialization and setup sequence that typically performs some or all of the following tasks:

- Stack pointer setting
- Memory Management Unit setup
- Enabling caches
- Enabling interrupts
- Clock PLL initialization and auto gating
- Selecting security mode

As tests are being developed and debugged or a regression is being run, this set of common initializations does not change as the figure below shows. In this section, we describe the implementation details of how simulation of tests can be launched, not from the start, but rather from where the specialized portion of the native test kicks in as shown in Figure 5.

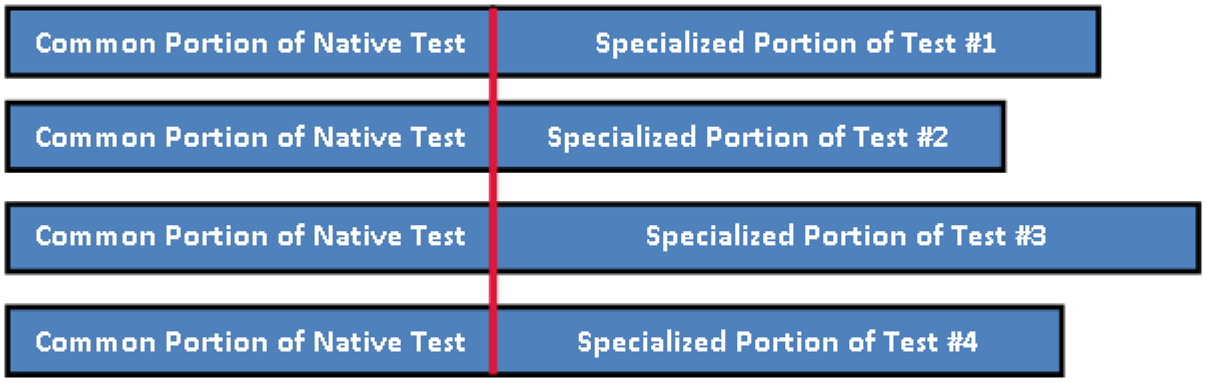

Common Portion of Native Test
Specialized Portion of Test #1
Common Portion of Native Test
Specialized Portion of Test #2
Common Portion of Native Test
Specialized Portion of Test #3
Common Portion of Native Test
Specialized Portion of Test #4

Figure 5 Identifying Common Execution Points amongst C Tests

5.1 How Native C Code is Partitioned and Where it Resides

As is always the case with implementing a Save/Restore flow, a well-defined and stable save point must be reached, where simulation can be paused to create a saved snapshot. It is not very straightforward to identify such a save point during processor execution as simulation is running. If not considered thoroughly, a slightly misguided save point may corrupt the processor state and other related variables after a specialized test image is loaded.

To delineate the common native code from specialized code, the former will branch to a **`master()`** routine, which starts the actual test.

```
void master( ) {
  // TEST CODE STARTS HERE
}
```

To avoid the concerns stated earlier, a dedicated region in the scatter file was set aside to hold the code that performs the main testing and is different for each test.

```
 MY_SECTION 0x34060000 0x7000 {
  *(MASTER_SECTION)
}
```

Figure 6 illustrates a typical processor memory map. In this example the **`master()`** routine of the scatter file is in the SRAM region of memory.

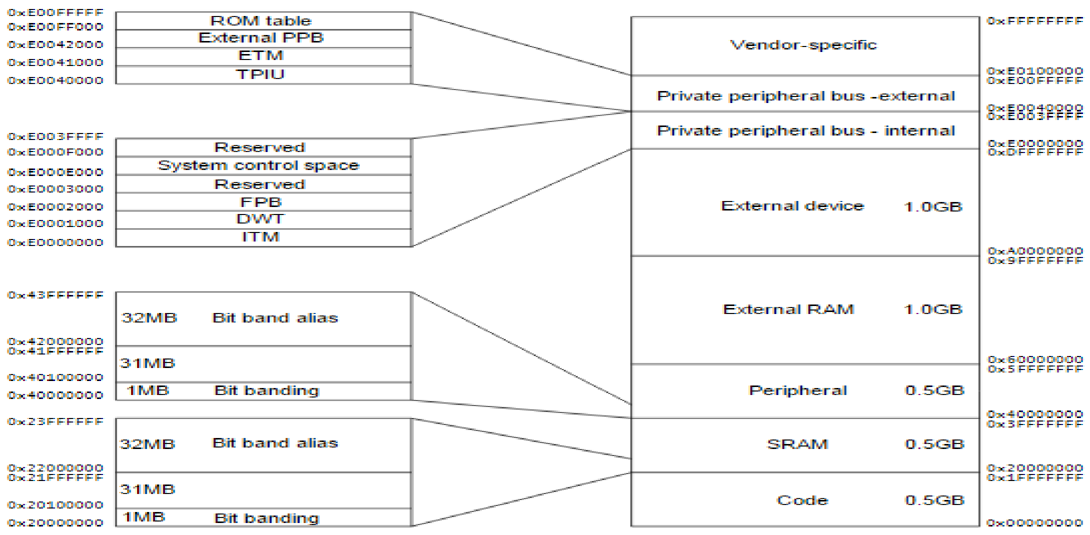


Figure 6 Processor's Memory Map and Location of Scatter File

5.2 How Synchronization Between C and Simulator is Achieved for Save Point

Once the common native code has finished, the simulator has to be signaled to stop executing and perform a checkpoint. When a call is made to the `master()` routine, the processor starts fetching its corresponding code using the routine's start address. This is the event that needs to be intercepted by the simulator. This signaling is facilitated by creating a conditional breakpoint using the `stop` command of VCS's TCL based command line interface (UCLI). The condition that is being monitored occurs when the processor fetch address matches the start address of the master routine. The `stop` command is versatile, and besides setting the condition, can also specify what commands to be executed when the condition is reached as implemented in the following TCL snippet. This is illustrated in Figure 7.

```
if { $env(SAVE)== 1} {
 stop  -name SaveBP
       -once
       -condition {/top/…/ARADDRM[39:0] == 40'h34067000}
       -continue
       -command {  puts "SAVING SNAP";
                   run 0;
                   save SNAP1;
                   puts "SAVED SNAP";
                   run
                 }
}
```

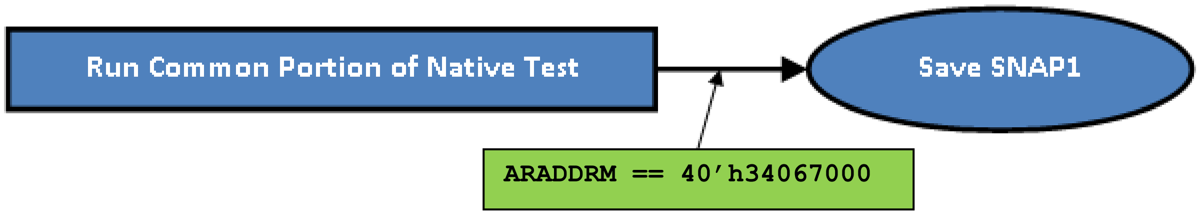


Figure 7 Identify the Event to Notify VCS to Create Snapshot

This breakpoint is set only when in the **save** mode of operation as it would not be needed during **restore**. It is given a user-defined name, SaveBP, so it can be referenced in the script if needed, for example, to disable or re-enable it. Once the specified condition is observed, the simulator is instructed to continue by executing the commands captured with the **`command`** option. Note that in order to achieve a stable point in the scheduling paradigm of the simulator, before checkpoint takes place, it is necessary to execute the **`run 0`** command, after which **`save`** can be executed. At this point, one can either run the test to the end or decide to exit. Either way, a snapshot has been generated only once and can be reloaded for the group of tests that have a common native startup.

Notice that we decided to use a command line interface to implement the Save/Restore flow. We felt this would provide more flexibility and also be less intrusive to our testbench code. However, similar capabilities can also be achieved if one decides to use the testbench flow, which was briefly discussed earlier in section 4.1.2.

5.3 *How Restore is Implemented*

As shown in Figure 8, during restore, to bypass the re-execution of the common code, the first command in the runtime TCL script restores the simulation to the saved state. Effectively, the startup happens close to zero time. To run a new C test, it first has to be compiled to create a memory image that resides in a file. This compiled image is what differentiates the tests from one another. Once the memory region of the MASTER_SECTION image is loaded using the read command in the run time script, simulation can be resumed from the saved point. This is the entry point to develop and debug the specifics of each C test without the burden of a lengthy startup time.

```
if { $env(RESTORE)== 1} {
  restore SNAP1
  puts "Loading new test code"
  memory -read "top/…/RAM/array"
         -file "MY_SECTION"
         -radix hex
         -start 32'h0000
  run
}
```

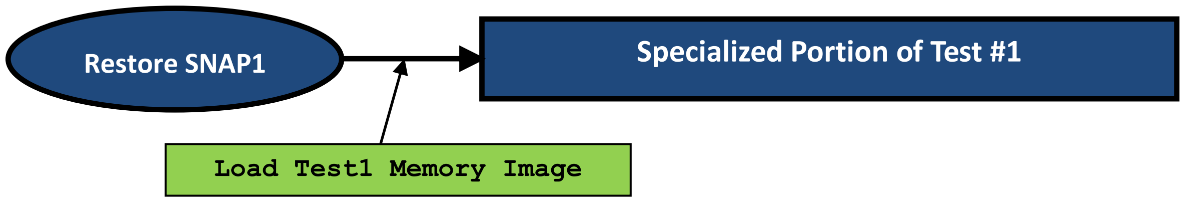


Figure 8 Restore a Saved Snapshot, Load New Test and Run

5.4 Best Practices

To implement the aforementioned Save/Restore capability most effectively, a solid understanding of the embedded testing environment is required. This is especially true if the flow has to intercept the simulation at various stages in the execution of the C test and create multiple save points.

- Embedded C test always needs to be loaded in separate memory section with use of scatter file.
- The simulation save point should be just before the test code is fetched by processor.

For seamless deployment of the flow, script automation is required in the regression environment to hide the implementation details. For example, SAVE and RESTORE variables can be used as:

- Call TCL routine to create save point
  ```
  make runtest TEST=Test1 USING=VCS64 SEED=1 SAVE=1
  ```

- Call TCL routine to restore simulation from saved point, load new C test in memory and start simulation
  ```
  make runtest TEST=Test1 USING=VCS64 SEED=1 RESTORE=1
  ```

For the latter example, the script should check if the characteristics of the test matches any of the saved snapshots in the library so it can be properly restored and if one doesn't exist, simulation needs to be run to create the necessary checkpoint.

We suggest that you consult and collaborate with your Synopsys AC, as there are recommended best practices from the tool's perspective, some of which are:

- TCL procedures that were sourced at the time of **save** will not be restored automatically and need to be explicitly sourced again after **restore**.
- Update your scripts to make sure the OS and patch versions of the **restore** host match those of the **save** host.
- Bring simulation to a stable point before **save** by executing the UCLI command **run 0**, to guarantee all pending events in current time slot are completed.
- For the same OS having Address Randomization, set the following variable:
  setenv VCS_ENABLE_ASLR_SUPPORT 1

6 Conclusions

Improving an existing infrastructure to cut down simulation turnaround time for large SoC's can be a major undertaking and the potential benefits should be substantial enough to justify the effort. Within a month after the implementation of the Save/Restore flow was started, we were able to have a basic prototype up and running. The initial focus was on only three tests sharing a boot up sequence, running on RTL. Due to our special application, a couple of tool issues were uncovered, for which Synopsys provided temporary TCL workarounds and were later fixed in VCS. We also experimented and came up with a suitable synchronization mechanism between C tests and VCS as described in the paper.

Once integrated in the flow, test developers reported time saving of 30 to 90 minutes per debug iteration since the common and already verified initialization sequence could be avoided to immediately get to the point of failure in the main C test. The FSDB dumping proved flexible to allow either auto-merging, split-dumping or dumping only after restore.
The same results were achieved at regression time when the same saved snapshot was reused for each one of the tests in a group that share the same C initialization. The 30-90 minute saving per test, aggregated to about 30-35% reduction in the overall regression time.

Since this methodology is agnostic to the type of simulation being run, other groups running power-aware (VCS-NLP) and SDF gate level simulations have reported similar benefits.

Going forward, we'd like to expand this flow to UVM tests for which UVM_TESTNAME changes, but the tests still share the same embedded-C boot up sequence.

7 References

[1] Verdi User Guide https://solvnet.synopsys.com/dow_retrieve/latest/ni/verdi.html

[2] VCS User Guide https://solvnet.synopsys.com/dow_retrieve/latest/ni/vcs_mx.html

[3] ARM Software Development Tools http://infocenter.arm.com/help/index.jsp