ABSTRACT

KULKARNI, KSHITIJ DWARKADHISH. UVM-based Verification Suite for a Cache. (Under the direction of Dr. W. Rhett Davis.)

This thesis describes the work done towards creating a verification environment for the data cache in H3 microprocessor using Universal Verification Methodology (UVM). UVM is a Accelera standard and combines features from Open Verification Methodology (OVM) and Verification Methodology Manual (VMM).

The cache was verified for functional behavior in [4]. In this thesis, we have concentrated on defining a comprehensive coverage metric and developing stimulus to achieve this coverage.

In the process of developing a verification suite for the cache, a simple testbench using SystemVerilog object-oriented programming (OOP) principles was converted to UVM. The UVM features used to do this and the steps used in modifying the testbench are documented. This serves to act as a guide to convert a simple testbench to a UVM compliant testbench. This exercise also helped in understanding the many advantages of using a well-defined methodology like UVM for developing verification suites over using only SystemVerilog constructs. The UVM features are compared against the simple OOP-testbench in terms of factors like the ease of developing the testbench structure, ease of developing new tests, reusability, etc.

For a target reader that may use the developed UVM-based verification suite to verify the cache, we've added a chapter describing the process of adding new tests to the verification suite.

Keywords: RTL Verification, Cache, UVM, OOP, testbench
UVM-based Verification Suite for a Cache

by

Kshitij Dwarkadhish Kulkarni

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

2016

APPROVED BY:

Dr. Eric Rotenberg

Dr. Paul D. Franzon

Dr. W. Rhett Davis
Chair of Advisory Committee
DEDICATION

Dedicated to my mother who has taken me through many rough times without me feeling any bumps along the way.
BIOGRAPHY


He joined NC State University in Fall 2014 to pursue Master Of Science in Computer Engineering.
ACKNOWLEDGEMENTS

Every work is a result of contributions from many people. I would like to take this opportunity to express my gratitude towards everyone who directly or indirectly helped me accomplish this goal.

I would like to thank my advisor Dr. W. Rhett Davis for encouraging me to take up this idea for my thesis. His clear directions about the goals of the project helped me stay focused. Otherwise I would've been, in his own words, "a deer stuck in headlights". His advice for writing the thesis made a difficult task simple. I'm grateful that he helped me get funding during my effort. I would also like to thank my committee members. Dr. Eric Rotenberg for making Computer Architecture interesting and Dr. Paul Franzon for his words "design before coding" which I always use as a mantra. I would like to thank Mentor Graphics for funding my efforts.

I would like to thank my other teachers at NC State, Dr. Huiyang Zhou for helping me generate an interest in Computer Architecture, Dr. Brian Floyd for helping me get over my phobia of transistors.

I would like to thank Rangeen Basu Roy Chowdhary and Vinesh Shrinivasan whose prior efforts I am working on. They were always there to help me understand the little details. Verification is a constant dialog between the verification engineer and the design engineer. I am grateful that Rangeen and Vinesh humored all my questions.

I would like to thank my teachers in India Dr. Narayan Marathe, Dr. Shaila Subbaraman, Dr. Y. V. Joshi, Dr. Ganesh Bhokare and Dr. M. Prakash for feeding my curiosity.

My family members, Dr. Geeta Kulkarni, Parikshit and Shrawni and all my extended family in India who have always wished the best for me. I would like to thank my many friends in India Neeraj, Yogesh, Sandesh, Aonkar, Swapnil who made themselves available for me whenever I needed them. I would like to thank Sumedha for being the sounding board for my complaints against everyone.

I would like to thank my family in Raleigh: my room-mates Anish, Omkar, Rohit for bearing with me when I was working on my thesis. Thanks Swati, Reshma, Chaitanya, Sourabh, Rutuja, Salil, Mihir, Pranit for being great friends and support.

Finally, I know he won't read this, but I would like to thank Louis C.K., comedian/story-teller/writer/pseudo-philosopher (in that order) for keeping me sane for the past 1 year.
# TABLE OF CONTENTS

LIST OF TABLES .................................................................................................................. viii

LIST OF FIGURES ................................................................................................................ ix

**Chapter 1 INTRODUCTION** .............................................................................................. 1
  1.1 Introduction .................................................................................................................... 1
  1.2 Outline .......................................................................................................................... 3

**Chapter 2 Cache Specification** .......................................................................................... 4
  2.1 Cache Specification ....................................................................................................... 4
  2.2 Store Features ............................................................................................................... 5
    2.2.1 Store Interface signals ........................................................................................... 5
    2.2.2 Store Requests ...................................................................................................... 6
    2.2.3 Store Buffer ......................................................................................................... 7
  2.3 Load Features ............................................................................................................... 8
    2.3.1 Load interface to and from the cache ..................................................................... 8
    2.3.2 Load request and responses: ................................................................................ 9
    2.3.3 Load sizes: .......................................................................................................... 9
    2.3.4 Non-blocking interfaces: ...................................................................................... 10
    2.3.5 Store-load forwarding: ....................................................................................... 10
    2.3.6 Cache fill request and response from lower hierarchy: ....................................... 11
    2.3.7 Miss Status Handling Register (MSHR): ............................................................ 11
    2.3.8 MSHR delays: .................................................................................................... 12
    2.3.9 MSHR data: ....................................................................................................... 12
  2.4 Invalidation requests: .................................................................................................... 13
  2.5 Scratch mode: ............................................................................................................. 13
    2.5.1 Scratch-mode interface signals: .......................................................................... 14
    2.5.2 Load interface for scratch-mode: ........................................................................ 14
    2.5.3 Store interface for scratch-mode: ....................................................................... 15

**Chapter 3 OOP-Testbench** .............................................................................................. 16
  3.1 Interface: ....................................................................................................................... 17
  3.2 Transactor: .................................................................................................................. 18
  3.3 Generating stimulus. tester: ....................................................................................... 20
  3.4 Applying stimulus. Driver: ......................................................................................... 21
  3.5 Tester to driver interface: ........................................................................................... 23
  3.6 Monitoring the responses from the DUT. Monitor: .................................................... 23
  3.7 Generating ideal responses. Scoreboard: ..................................................................... 25
  3.8 Comparing the response. Checker: ............................................................................. 26
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2.1</td>
<td>Core-to-Cache Store Interface Signals</td>
<td>5</td>
</tr>
<tr>
<td>Table 2.2</td>
<td>Store Sizes</td>
<td>5</td>
</tr>
<tr>
<td>Table 2.3</td>
<td>Cache-to-memory. Store interface signals</td>
<td>6</td>
</tr>
<tr>
<td>Table 2.4</td>
<td>Core-to-cache. Load interface signals</td>
<td>8</td>
</tr>
<tr>
<td>Table 2.5</td>
<td>Load sizes</td>
<td>8</td>
</tr>
<tr>
<td>Table 2.6</td>
<td>mem2cache interface signals</td>
<td>9</td>
</tr>
<tr>
<td>Table 2.7</td>
<td>Scratch Mode Interface signals</td>
<td>14</td>
</tr>
<tr>
<td>Table 5.1</td>
<td>core_cache_if class details</td>
<td>65</td>
</tr>
<tr>
<td>Table 7.1</td>
<td>Comparison for adding tests</td>
<td>90</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.1</td>
<td>OOP-Testbench Structure</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>Transactor</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>Tester run() task</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>Driver run() task</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>core_cache_if class</td>
<td>24</td>
</tr>
<tr>
<td>3.6</td>
<td>Monitor run() task</td>
<td>25</td>
</tr>
<tr>
<td>3.7</td>
<td>Testbench run() task</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>UVM Factory</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>UVM Factory in action</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>Registering Object with Factory</td>
<td>33</td>
</tr>
<tr>
<td>4.4</td>
<td>create() function to instantiate objects</td>
<td>33</td>
</tr>
<tr>
<td>4.5</td>
<td>Storing in uvm_config_db</td>
<td>34</td>
</tr>
<tr>
<td>4.6</td>
<td>Retrieving from uvm_config_db</td>
<td>34</td>
</tr>
<tr>
<td>4.7</td>
<td>constructor and build_phase</td>
<td>36</td>
</tr>
<tr>
<td>4.8</td>
<td>Raising Objections</td>
<td>37</td>
</tr>
<tr>
<td>4.9</td>
<td>Connections using multiple mailboxes</td>
<td>38</td>
</tr>
<tr>
<td>4.10</td>
<td>Observer Design Pattern connections</td>
<td>39</td>
</tr>
<tr>
<td>4.11</td>
<td>Register Monitor with Factory</td>
<td>44</td>
</tr>
<tr>
<td>4.12</td>
<td>UVM Analysis Ports in Monitor</td>
<td>45</td>
</tr>
<tr>
<td>4.13</td>
<td>Monitor build_phase()</td>
<td>45</td>
</tr>
<tr>
<td>4.14</td>
<td>Monitor</td>
<td>46</td>
</tr>
<tr>
<td>4.15</td>
<td>uvm_tlm_analysis_fifos in Checker</td>
<td>46</td>
</tr>
<tr>
<td>4.16</td>
<td>Checker build_phase()</td>
<td>47</td>
</tr>
<tr>
<td>4.17</td>
<td>Checker run_phase()</td>
<td>47</td>
</tr>
<tr>
<td>4.18</td>
<td>Checker</td>
<td>48</td>
</tr>
<tr>
<td>4.19</td>
<td>Register Scoreboard with Factory</td>
<td>48</td>
</tr>
<tr>
<td>4.20</td>
<td>Scoreboard</td>
<td>49</td>
</tr>
<tr>
<td>4.21</td>
<td>Coverage</td>
<td>50</td>
</tr>
<tr>
<td>4.22</td>
<td>Extend core_cache_if from uvm_sequence_item</td>
<td>51</td>
</tr>
<tr>
<td>4.23</td>
<td>Driver class definition</td>
<td>51</td>
</tr>
<tr>
<td>4.24</td>
<td>Driver run_phase()</td>
<td>52</td>
</tr>
<tr>
<td>4.25</td>
<td>Driver</td>
<td>53</td>
</tr>
<tr>
<td>4.26</td>
<td>Agent build_phase()</td>
<td>54</td>
</tr>
<tr>
<td>4.27</td>
<td>Agent run_phase()</td>
<td>54</td>
</tr>
<tr>
<td>4.28</td>
<td>Agent</td>
<td>55</td>
</tr>
<tr>
<td>4.29</td>
<td>Env</td>
<td>56</td>
</tr>
<tr>
<td>4.30</td>
<td>top module initial() block</td>
<td>57</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>base_test: Define env and sequencer</td>
<td>60</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>base_test: build_phase</td>
<td>60</td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>base_test: end_of_elaboration_phase</td>
<td>61</td>
</tr>
<tr>
<td>Figure 5.4</td>
<td>base_test sequencer connections</td>
<td>61</td>
</tr>
<tr>
<td>Figure 5.5</td>
<td>sequence class definition</td>
<td>62</td>
</tr>
<tr>
<td>Figure 5.6</td>
<td>body() task example</td>
<td>63</td>
</tr>
<tr>
<td>Figure 5.7</td>
<td>Sequences using other sequences</td>
<td>64</td>
</tr>
<tr>
<td>Figure 5.8</td>
<td>run_phase() of a test</td>
<td>68</td>
</tr>
<tr>
<td>Figure 5.9</td>
<td>Multiple sequences in parallel</td>
<td>69</td>
</tr>
<tr>
<td>Figure 5.10</td>
<td>UVM_TEST</td>
<td>70</td>
</tr>
<tr>
<td>Figure 5.11</td>
<td>dcache_tb_package</td>
<td>71</td>
</tr>
<tr>
<td>Figure 7.1</td>
<td>Load request to same line, different tag. Request Forwarded</td>
<td>93</td>
</tr>
<tr>
<td>Figure 7.2</td>
<td>Load request to same line, different tag. Request not forwarded</td>
<td>94</td>
</tr>
</tbody>
</table>
CHAPTER

1

INTRODUCTION

1.1 Introduction

Digital systems are getting larger in size (in terms of number of transistors) and complexity. Multi-core processors are now much more common and the design complexity involved in them increases the verification effort significantly. Multiple cores add the additional design and verification effort of cache-coherence to them. Constrained-random tests are necessary to verify any design of this complexity. Moreover since the turnaround time for new designs is expected to reduce, re-usability of verification environments is critical. Industry bodies, like Accellera, have developed many methodologies that help in developing re-usable testbenches. We'll like to explore the Universal Verification Methodology in this thesis. UVM is an extensive base-class library to develop verification environments. It is now an IEEE standard for verification.

The H3 processor[4] is a heterogenous multi-core processor. It consists of two processor
cores, a one-wide core and a two-wide core. The cache designed for the processor has a simple invalidation-only coherence protocol for maintaining coherence between the cores. It also has cache-core decoupling for ease of thread-migration[4]. Our focus will be exploring the process of verification of this cache using the Universal Verification Methodology (UVM).

More about UVM: it is a base-class library over the existing SystemVerilog verification language. It was standardized by Accelera through efforts from EDA vendors and contributions from established methodologies like Open Verification Methodology (OVM) and Verification Methodology Manual (VMM)[2][5]. The UVM features include methods of connecting different components of a testbench with clearly defined interfaces which helps in maintaining the testbench over multiple iterations of a project. The idea is that while the internals of each components change, the interfaces shouldn't. Hence the structure becomes easier to modify and maintain.

This cache was previously verified for functional purposes[4]. This thesis describes the verification of the cache to cover all the features of the cache, some of which may not have been covered during functional testing. We define a coverage goal and try to achieve it by defining stimulus. Before doing that we need to document the specifications of the cache so that we can define the coverage better. In satisfying the above goal, we hope to explore the features of UVM and document the various advantages that we gain by doing so. To do that we need to have a baseline testbench to compare it to. We start with a testbench developed using the Object Oriented features of the SystemVerilog language. This testbench is very similar to the one developed for ECE745: ASIC Verification class taught at NC State University. Then we explain the procedure used to convert this testbench to a UVM-compliant testbench.

Summarizing our major goals:

1. Documenting the specification of the H3 cache.

2. Developing a baseline SystemVerilog based Object-Oriented testbench (OOP-testbench) for the cache.

3. Converting the OOP-testbench to a UVM-based verification-suite for the cache.

4. Documenting the process of modifying the OOP-testbench developed in SystemVerilog to make it UVM compatible.
5. Defining the coverage metric for the cache.

6. Exploring the advantages that we gained by using the UVM over the OOP-testbench.

### 1.2 Outline

The document is divided into 8 chapters. In chapter 2, we formally describe the cache features and specifications against which we will be verifying the cache. Chapter 3 and 4 explore the structure of the testbench. In chapter 3 we look at a testbench for the cache developed using basic Object-Oriented Programming principles of SystemVerilog. In chapter 4 we will be looking at the UVM features and will be using them by modifying our existing OOP-testbench. In this way, we not only hope that the reader will get an idea of UVM, but also will get a step-by-step guideline of how to convert a OOP-testbench to be UVM compatible. In chapter 5 we explain how to add new tests to the UVM verification suite. This is targeted towards users who will be using the verification-suite to verify the cache in future projects. In chapter 6, we concentrate on the coverage metric for the verification effort and the stimulus used to achieve the coverage. Chapter 7 discusses the results of the effort. We include our concluding remarks in chapter 8.
Before discussing the testbench and the coverage metric, we spend some time looking at the specifications of the cache. This will give us better understanding of the features and the way they interact. Internal details of the design-under-test (DUT) functionality are also important when measuring coverage. For example, if the DUT has a state machine, we need to ensure that we are traversing all the valid transitions of the state machine.

2.1 Cache Specification

The cache is a direct-mapped, write-through write-not-allocate cache. The line-size, number of lines are configurable and the verification suite keeps them configurable. But for the cache used in H3 processor:

1. Cache-size: 2 kB
2. Line-size: 128 bits
2.2 Store Features

2.2.1 Store Interface signals

- Following are the store interface signals between the cache and the upper hierarchy. Each signal’s name ends with the direction it has been indicated.

Table 2.1 Core-to-Cache Store Interface Signals

<table>
<thead>
<tr>
<th>Signal</th>
<th>Size</th>
<th>Active high</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>stEn_i</td>
<td>1 bit</td>
</tr>
<tr>
<td>Input</td>
<td>stAddr_i</td>
<td>32 bits</td>
</tr>
<tr>
<td>Input</td>
<td>stSize_i</td>
<td>2 bit</td>
</tr>
<tr>
<td>Input</td>
<td>stData_i</td>
<td>32 bits</td>
</tr>
<tr>
<td>Output</td>
<td>stHit_o</td>
<td>1 bit</td>
</tr>
<tr>
<td>Output</td>
<td>stMiss_o</td>
<td>1 bit</td>
</tr>
<tr>
<td>Output</td>
<td>stbEmpty_o</td>
<td>1 bit</td>
</tr>
<tr>
<td>Output</td>
<td>stallStCommit_o</td>
<td>1 bit</td>
</tr>
</tbody>
</table>

- The different sizes that the store can have are shown in table below.

Table 2.2 Store Sizes

<table>
<thead>
<tr>
<th>Value</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Byte</td>
</tr>
<tr>
<td>01</td>
<td>Half-Word</td>
</tr>
<tr>
<td>01</td>
<td>Word</td>
</tr>
</tbody>
</table>

- The signals in the store interface between the cache and the lower hierarchy follow a specific naming scheme. They start with the initiator and target module. For example,
2.2. STORE FEATURES

Table 2.3 Cache-to-memory. Store interface signals

<table>
<thead>
<tr>
<th></th>
<th>Signal</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>dc2memStAddr_o</td>
<td>30 bits</td>
</tr>
<tr>
<td>Output</td>
<td>dc2memStData_o</td>
<td>32 bits</td>
</tr>
<tr>
<td>Output</td>
<td>dc2memStByteEn_o</td>
<td>4 bits</td>
</tr>
<tr>
<td>Output</td>
<td>dc2memStValid_o</td>
<td>1 bit</td>
</tr>
<tr>
<td>Input</td>
<td>mem2dcStComplete_i</td>
<td>1 bit</td>
</tr>
<tr>
<td>Input</td>
<td>mem2dcStStall_i</td>
<td>1 bit</td>
</tr>
</tbody>
</table>

"dc2mem" implies data cache to memory, where memory is a substitution for lower hierarchy of the memory. The signals are explained in table 2.3.

2.2.2 Store Requests

- The upper hierarchy sends a store requests using the stEn_i, stAddr_i, stSize_i and stData_i signals.

- Stores can have different sizes as shown in the table.

- The store requests should only be sent to the cache if the store buffer has space. The stallStoreCommit_o signal is used to signal to the core/upper-hierarchy that the store buffer is full.

- The cache forwards the store requests received from the upper-hierarchy in order that they are received. It sends using the dc2memStAddr_o, dc2memStData_o, dc2memStByteEn_o and dc2memStValid_o signals.

- Since the stores can have different sizes, but the store data signal is 32 bits, we maintain the dc2memStByteEn_o signal to indicate which bytes in the sent word are valid. We have one bit in the dc2memStByteEn_o signal corresponding to each byte in a word.

- The lower hierarchy responds to a store with a store complete signal when done. It does so with the mem2dcStComplete_i signal.
2.2. STORE FEATURES

• Once the response is received, we check if the address is present in the cache line. If it is we respond to the upper-hierarchy with a hit, if not we respond with a miss.

• The delay between the upper-hierarchy sending a store request and the cache responding with a hit/miss response can be large and isn't fixed.

• If the store address is present in the cache lines, it is necessary to merge the store data with the cache line data to update it.

• If the address isn't present in the cache line, but is present in the MSHR, we merge the store data with the MSHR data. More details on the MSHR data are mentioned in section 2.3.9.

• Once the store complete response is received and the data has been merged with the cache line or MSHR data, the store is considered to be committed.

2.2.3 Store Buffer

• The store buffer stores the store requests that have been sent to the lower hierarchy, but haven't been committed yet. It is maintained as a circular FIFO.

• New store commands are pushed to the tail of the buffer. The buffer stores the data, the address, the size and the byte enable signal of the store command. These values are used when the store is committed.

• The buffer can store up to 8 entries, so up to 8 stores can be outstanding at any given time.

• The responses to the store are received in a sequential way. The store complete response from the lower-hierarchy doesn't have any additional address/size information with it. But we can safely conclude that each store complete response is done in order and it corresponds to the current head of the store buffer since that is the oldest store that is pending.

• Once the response is received and the store is committed, we remove the command from the store buffer and increment the head.
2.3 Load Features

2.3.1 Load interface to and from the cache

- The cache load input from the upper hierarchy are as shown in Table 2.4. The cache supports loads with different sizes. The naming scheme remains similar to the one used in store signals.

Table 2.4 Core-to-cache. Load interface signals

<table>
<thead>
<tr>
<th>Signal</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldEn_i</td>
<td>1 bit</td>
</tr>
<tr>
<td>ldAddr_i</td>
<td>32 bits</td>
</tr>
<tr>
<td>ldSize_i</td>
<td>2 bits</td>
</tr>
<tr>
<td>ldData_o</td>
<td>32 bits</td>
</tr>
<tr>
<td>ldHit_o</td>
<td>1 bit</td>
</tr>
<tr>
<td>ldMiss_o</td>
<td>1 bit</td>
</tr>
<tr>
<td>ldDataValid_o</td>
<td>1 bit</td>
</tr>
</tbody>
</table>

- The different valid sizes are shown in table 2.5.

Table 2.5 Load sizes

<table>
<thead>
<tr>
<th>Value</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Byte</td>
</tr>
<tr>
<td>01</td>
<td>Half-Word</td>
</tr>
<tr>
<td>10</td>
<td>Word</td>
</tr>
</tbody>
</table>

- The cache load interface to and from the lower hierarchy are shown in table 2.6. The
2.3. LOAD FEATURES

naming scheme "dc2mem" and "mem2dc" signify the initiator and target module in that order. For example dc2mem specifies that the initiator is data cache and target is memory.

Table 2.6  mem2cache interface signals

<table>
<thead>
<tr>
<th>Signal</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output dc2memLdValid_o</td>
<td>1 bit</td>
</tr>
<tr>
<td>Output dc2memLdAddr_o</td>
<td>28 bits</td>
</tr>
<tr>
<td>Input mem2dcLdCpkt_i</td>
<td>2 bits</td>
</tr>
<tr>
<td>Input mem2dcLdTag_i</td>
<td>21 bits</td>
</tr>
<tr>
<td>Input mem2dcLdData_i</td>
<td>128 bits</td>
</tr>
<tr>
<td>Input mem2dcLdValid_i</td>
<td>1 bit</td>
</tr>
</tbody>
</table>

2.3.2 Load request and responses:

- The upper hierarchy sends a load requests using the ldEn_i, ldAddr_i, ldSize_i signals.

- The responses to load request coming from the upper hierarchy should be given in the same cycle, i.e., hit/miss should be valid at the next clock edge. Each load request should result in either one of the hit or miss signals to be valid.

- If the result is a load hit, the data and the data valid signals should also be valid at the next clock edge. The ldData_o, ldHit_o, ldMiss_o and ldDatavalid_o signals are used to send the response.

2.3.3 Load sizes:

- Cache supports load requests of different sizes.

- The load data output sent using the ldData_o for each size will be aligned to the LSB irrespective of the size. For example, a half-word data will be aligned to the least
significant half-word and rest of the bits will be zero; a byte will be aligned to the LSB and rest bits will be zero.

2.3.4 Non-blocking interfaces:

• The cache interfaces are non-blocking. It can accept requests even after suffering a cache miss from the upper hierarchy and doesn't block any interactions with the lower hierarchy.

• The cache has one Miss Status Handling Register. It can handle and re-fill a single load miss at a time. But as mentioned earlier, it can still handle new requests from the upper-hierarchy when pending on a cache-miss and can send hit/miss responses. But it cannot handle more than one cache-refills at a time.

2.3.5 Store-load forwarding:

• The cache implements store-load forwarding. If a data requested by the load is present in the cache line and in the store buffer, the data in the store buffer is forwarded since it is more recent than the one in the cache line.

• The same address can be stored at many positions in the store-buffer owing to multiple store requests to the same address. We always chose the most recent entry to forward.

• Different sizes of the load and store makes store load forwarding complicated. We only forward data in the store buffer if the load size requested is less than or equal to the store size in the store buffer. We also ensure that the addresses are perfectly aligned. This is done to simplify the implementation of store-load forwarding. Such cases, where the requested data is only partially present or isn't properly aligned, are referred to as "partial hits". In the case of partial hits, not only do we not forward the data from the store buffer, we cannot send the data stored in the cache line since it would be stale. If the most recent hit in the store-buffer is a partial hit, we have to signal a miss, even if the address is present in the cache-line.
2.3.6  Cache fill request and response from lower hierarchy:

- The cache can forward a load request to the lower hierarchy if it misses in the cache and the MSHR is empty. The dc2memLdAddr_o and dc2memLdValid_o signals are used to send this request.

- The lower hierarchy responds with a 128 bit data line. The mem2dcLdCpkt_i, mem2dcLdTag_i, mem2dcLdIndex_i, mem2dcLdData_i and mem2dcLdvalid_i signals are used for this.

- The mem2dcLdCpkt_i signal stands for load command packet signal. The commands from lower hierarchy can be used to fill a cache line or to invalidate a cache line. Command packet signal is used to differentiate between the two. The command for a cache line fill is 0x1, that for invalidate is 0x2.

- The cache fill request should be sent only when the address isn't present in the cache line. A miss resulting due to a partial hit when the data is already present in the cache shouldn't be forwarded to the lower-hierarchy.

2.3.7  Miss Status Handling Register (MSHR):

- The cache has one MSHR.

- The MSHR is used to store the state of a cache fill request sent to the lower hierarchy. The cache can send such a request only if it has enough space in the MSHR to store the state of the request. Consequently since we have only one MSHR, we can only have one outstanding fill request at a time.

- Any further cache misses that occur after one fill request has been sent are not scheduled for fill till the first request is satisfied.

- Data that is received from the lower hierarchy is matched against the MSHR to ensure that we are receiving the correct address. Only if the address for the response matches the one in the MSHR, we accept the data and fill the cache-line with it.
2.3. LOAD FEATURES

• It is required by the upper hierarchy to keep on retrying for a hit, the cache doesn’t give a notification to upper-hierarchy if the outstanding fill request was completed.

• The MSHR also has the feature of storing the store requests that went through the cache for the address that the MSHR stores. We’ll discuss this feature in a different section.

2.3.8 MSHR delays:

• If the cache suffers a miss in the cache line and the MSHR has space for the new request, the following is the sequence of events:
  – Clock cycle N: Load request, misses in the cache line and the MSHR isn’t valid.
  – Clock cycle N+1: Cache-fill request is sent to the lower hierarchy. The MSHR is filled with the address.

• If the lower hierarchy responds with the data, the sequence of events is as follows:
  – Clock cycle N: Cache-fill Data received from the lower hierarchy corresponding to the MSHR state.
  – Clock cycle N+1: Cache-line is filled, MSHR is cleared.
  – The data filled in the cache line can only be valid in cycle N+2.
  – Although the MSHR also is cleared and valid to receive a new request in cycle N+2, the cache has flags to let it know that the MSHR is going to be cleared the next cycle. So, if a load request from upper hierarchy is received in cycle N+1, in cycle N+2, the MSHR will be filled with the new address and a load request is sent to the lower-hierarchy.

2.3.9 MSHR data:

• Once a load request has missed and a cache-fill request for the address is sent to the lower hierarchy, the address is stored in the MSHR. It is possible for the upper hierarchy to send store requests to addresses in the same cache line. These store requests are buffered in the store buffer.
2.4. INVALIDATION REQUESTS: CHAPTER 2. CACHE SPECIFICATION

- We have seen that when a store request commits, we merge the data with the cache-line if the address exists in a cache-line.

- For the address corresponding to the MSHR state, the address is going to be filled in a cache-line soon. If we have stores committing to the same cache line, we store the data in a "MSHR data" structure.

- This data is then merged with the incoming data when it is received from the lower hierarchy.

**2.4 Invalidation requests:**

- The lower hierarchy of the memory sends invalidation requests to the cache using the same interface that is used to send cache-fills. The signals used are mem2dcLdCpkt_i, mem2dcLdIndex_i, mem2dcLdValid_i.

- The other signals of the load interface mem2dcLdData_i and mem2dcLdTag_i aren't used for invalidation.

- This is because the tag field isn't used to check for invalidation. The lower memory hierarchy keeps a directory to indicate which addresses are stored in the cache. The directory will take care not to send an invalidation request if the target address isn't present in the cache.

- The value of load command packet for invalidation is 0x2.

**2.5 Scratch mode:**

In the Scratch mode, the cache acts like a SRAM. There is no load miss or store miss. Every command results in a hit. That is why it is important to ensure that every address that we access is pre-initialized to some value.
2.5. **SCRATCH MODE:**

### 2.5.1 Scratch-mode interface signals:

The scratch-mode signals are shown in table 2.7.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcScratchModeEn_i</td>
<td>1 bit</td>
</tr>
<tr>
<td>dcScratchWrAddr_i</td>
<td>11 bits</td>
</tr>
<tr>
<td>dcScratchWrEn_i</td>
<td>1 bit</td>
</tr>
<tr>
<td>dcScratchWrData_i</td>
<td>8 bits</td>
</tr>
<tr>
<td>dcScratchRdData_o</td>
<td>8 bits</td>
</tr>
</tbody>
</table>

- As can be noted, the load and store interfaces had a 32 bit data bus and a choice of selecting different sizes. The scratch interface on the other hand has a resolution of 1 byte.
- We don't check tags in the scratch mode, so the write address is only 11 bits long, i.e., index bits and the byte offset.
- The data on dcScratchWrData_i is written to the cache address dcScratchWrAddr_i when dcScratchWrEn_i and dcScratchModeEn_i is high on the next clock edge.
- The dcScratchRdData_o signal has the data stored at the dcScratchWrAddr_i as output.

### 2.5.2 Load interface for scratch-mode:

- The load interface stays mostly the same for scratch-mode. We use the same signals for a load access and to signal the load output. But, the internal functionality changes.
- We are said to be in scratch-mode if dcScratchModeEn_i was high in the previous cycle.
• In scratch-mode, if upper hierarchy sends a load request, the cache forwards the data present in the cache line irrespective of the value of cache valid bit and the tag value. In short, every load access results in a hit.

• Since, there isn't a miss, no cache-fill request is sent to the lower hierarchy.

2.5.3 Store interface for scratch-mode:

• Store interface stays almost the same. But the internal functionality changes.

• Instead of storing the command in the store buffer and waiting for the lower memory response, the cache commits the store immediately to the cache line.

• The cache doesn't send the store to the lower hierarchy, it doesn't check if the address is actually present in the cache. The result is always a store hit.

In the next chapter we discuss the structure of the testbench developed using SystemVerilog Object-Oriented Programming principles called the OOP-testbench.
The SystemVerilog language has many features in itself to make a verification environment modular and reusable. It has classes, interfaces, tasks, synchronization elements, coverage collecting constructs, etc. The use of classes makes the SystemVerilog an Object-Oriented hardware verification language. In this chapter we’ll describe a testbench that follows simple rules of the SystemVerilog language. We’ll call this a OOP-testbench[3] because it uses the object-oriented programming constructs of SystemVerilog that help in making the testbench modular. This will act as a baseline for comparison to the UVM-based testbench. We will quantitatively measure the effort needed to add tests or features to the OOP-testbench and compare it to the effort required for the same tasks in the UVM-based testbench.

The basic principle behind development of the testbench for a complex design is to separate the many tasks into components. This enables the same testbench to be used for different kinds of tests including directed as well as constrained-random tests. It allows small-changes to be made to each module as long as the interfaces between different modules are the same. The basic functionality of any testbench can be divided as[1]:
• Generating stimulus

• Applying stimulus to the DUT

• Monitoring the responses from the DUT

• Generating ideal responses for the generated stimulus

• Comparing the ideal response to the DUT response

• Gathering coverage information on the generated stimulus

Each of the above functionality should be executed by different components. These components should be separated from each other to make the structure reusable. Each of the testbench component is then divided into classes. Each class could possibly have its own thread to manage its functionality internally. This makes managing the data and the processes very simple. In the following sections of the chapter we'll look at each of testbench tasks and discuss which class takes care of each of these tasks.

Once these classes are defined along with the data and threads they encapsulate, they need to be connected to each other. SystemVerilog provides mailboxes for connections between different threads. As discussed earlier the interfaces need to be well-defined to make the testbench re-usable. The interfaces between these connections would be the class objects that will be passed using the mailboxes. Figure 3.1 shows the general structure of a OOP-testbench.

Before going into the details of the existing OOP-testbench structure, we'll first delve into the SystemVerilog interface that we've used for the testbench and the transactor construct which will be used extensively in the testbench.

3.1 Interface:

We've used the SystemVerilog interface construct to connect components of the DUT to the testbench. Since the testbench needs to apply stimulus and verify the responses, we need some way of providing the signal-level information of the DUT to the components of the testbench. The interface is used to do this. For the purpose of this testbench, the interface
is a place where we have bundled all the inputs/outputs of the DUT in one place. Instead of passing all the signals between the components, we need to only pass the interface. This makes adding, removing, modifying signals easier.

**3.2 Transactor:**

There are multiple definitions of a transactor. For our purpose a transactor is a class which has a specific structure. It is a construct which can have multiple threads. Each thread can have a single source (usually a mailbox or similar for SystemVerilog) from which it gets data in the forms of packets. The thread processes the packet of data, maybe generates its own packet based on the input packet and has a single destination (destination mailbox) to which it sends the output packet.

The transactor has a main thread which spawns all the other threads in the class. It can have other threads which don't have mailboxes, but support the transactor's functionality. But a single thread usually takes care of spawning all the other threads. A transactor structure is shown in Figure 3.2.

The main thread is usually called the run() task. A component using a transactor need only be aware of this run() task and spawn it. The transactor's run() task will take of spawning all the other threads.
Figure 3.2 Transactor
3.3 Generating stimulus. tester:

The testbench should generate stimulus corresponding to the test that needs to be run on the DUT. Since, we need to run a wide range of tests, the stimulus will change. So we should keep generating and applying stimulus separate from the rest of the structure of the testbench. Moreover, "generating" stimulus should be kept in a different class from "applying" stimulus to the DUT. There are multiple reasons for doing this.

First, applying stimulus will be done by changing input signals of the DUT. As the signals can change in the future, the changes due to changing signal definitions should be localized to a single class.

Second, we would like our stimulus and the testbench in general to be at the transaction-level. Rather than dealing with all the signal-level details, we would like to deal with the input stimulus and generated responses as transactions. As we will see, transaction-level modelling of the DUT is much easier than making a cycle-accurate model of the DUT.

Third, the generated signal itself can be complex and may need many layers to it. For example, for a networking application, we may have to generate layered packets as stimulus and check if errors in each layer are handled properly by the DUT. Hence the stimulus generation itself can be divided into multiple classes. The generator class is what deals with the generation of stimulus and passing the transaction to the driver class which takes care of applying the stimulus to the DUT. As can be seen from Figure 3.1 we have the test class also as part of the stimulus. The test is shown as an external component to the testbench because it changes for every test we want to run.

The generator can be designed as a transactor. In its run task it receives information from the test, create its own transaction packets and sends them to the driver. It can be thought of that the generator creates the input packets and the test class provides it with specific directions to generate these packets. The test can thus direct the generator to create packets that form a sequence of input transactions to achieve the test objective. In our design, the tester class acts as the generator. It has a single thread where it creates the transactions and sends them to the driver through a mailbox. It creates the transactions through a function called get_transaction(). The run() task takes an integer argument to specify how many packets the tester should create. The run() task is shown in Figure 3.3.
3.4 Applying stimulus. Driver:

The driver receives the transaction objects from the generator and converts them to pin wiggles on the DUT interface. The driver needs a copy of the DUT interface to get access to the DUT signals. The interface is passed to it as an argument when the driver is instantiated. It is important to note that in general the inputs are always applied just after the clock-edge so that they are valid at the next clock-edge.

The driver is also a transactor, waiting on the generator mailbox for packets. On receiving a packet, they are deciphered in terms of values to be applied to each signal. The driver then applies these values to the DUT interface signals.

In our testbench, we’ve decided to model all the interfaces between the cache and the outside world in the testbench itself. The cache has two interfaces: one to the upper-hierarchy/processor core and another to lower-hierarchy/memory. We need to model both

```verilog
    task run(input int n);
    `ifdef TESTER.PRINTS
        $display("Inside tester run. n: ", n);
    `endif
    repeat(n)
      begin
        send_transaction = get_transaction();

        core_2_cache_mbox.put(send_transaction);
      `ifdef TESTER.PRINTS
        $display($time, " Tester: put transaction in mailbox");
      `endif
    end // repeat (n)
    #100 $finish;
endtask // run
```

Figure 3.3 Tester run() task

The get_transaction() function is overridden by the specific test that is currently executing. The function directs the type of transactions to be generated. This function acts as the communication between the test class and tester.

3.4 Applying stimulus. Driver:
Currently we generate only upper-hierarchy input packets in the tester. The lower-hierarchy inputs are generated in the driver itself using the default $urandom functions. We have four different threads to handle all the functionality:

1. Core thread: Waits on the generator mailbox. If the received packet is a core input, apply it to the core input signals.

2. Memory thread: Get memory inputs from the core thread and store them in a queue to be used later. Monitor the DUT interface to detect either a load or store request going to the memory. Once a request is detected, notify the Load or Store thread about it.

3. Load thread: If a load request is received, wait for a random amount of time and send randomized data to the load input of the DUT.

4. Store thread: If a store request is received, wait for a random amount of time and send store complete transaction to the DUT.

The run task of the driver is shown in Figure 3.4. It forks off 4 threads that we described earlier.

```
  task run();
  `ifdef DRIVER_PRINTS
      $display("Inside driver run");
  `endif
  fork
    run_core();
    run_mem();
    run_load();
    run_store();
  join
  endtask // run
```

Figure 3.4 Driver run() task
3.5 Tester to driver interface:

The interface between the tester and the driver is a class which is used to send the input data through the mailboxes connecting them. In our design the class is called core_cache_if. It is in the cache_core.sv file of the code distribution. Figure 3.5 is a snippet of the class. Many of its internal variables are defined as random so that we can use the default randomize function of SystemVerilog to generate randomized transactions. The constraints are defined so that the inputs stay valid according to the cache specifications. These constraints are necessary since we would not want our randomized stimulus to be an invalid input to the design-under-test, unless we are testing for error conditions in the inputs.

3.6 Monitoring the responses from the DUT. Monitor:

The DUT will respond to the stimulus that we apply to its inputs. As we are modeling both the upper-hierarchy and lower-hierarchy interfaces, we need to test both these interfaces. The upper-hierarchy interface is fundamentally different from the lower-hierarchy interface because many outputs (load hit, load miss, load data valid, etc.) are asynchronous. We have made a design choice of monitoring the output of the two interfaces separately. This reduces the complication in monitoring and checking the two interfaces. These output transactions are sent to the checker class.

We have also made a design choice of monitoring the input signals on the interface and sending them to the scoreboard, rather than sending the generator packet to both driver and scoreboard. While the generator packet itself can be sent to the scoreboard, it adds complications to the scoreboard. The generator transaction may contain details like delays which the driver class decodes and sends the signals to the hardware-interface. The scoreboard shouldn't have to handle the same things again, since it is a transaction-level class. Moreover, the generator only generates the upper-hierarchy/core input and the memory input is generated by the driver. So it is important for us to create the input transactions in the monitor and send them to the scoreboard. The input transaction isn't divided into two interfaces like the output transactions are, we still handle them together.

Since we are trying to keep all our testbench components at the transaction-level, the
class core_cache_if;

rand bit load_en;
//rand bit [`SIZE_PC-1:0] load_address;
rand bit [`DCACHE_BYTES_IN_LINE_LOG-1:0] load_address_offset;
rand bit [`DCACHE_INDEX_BITS-1:0] load_address_index;
rand bit [`DCACHE_TAG_BITS-1:0] load_address_tag;
rand bit [`LDST_TYPES_LOG-1:0] load_size;

rand bit store_en;
//rand bit [`SIZE_PC-1:0] store_address;
rand bit [`DCACHE_BYTES_IN_LINE_LOG-1:0] store_address_offset;
rand bit [`DCACHE_INDEX_BITS-1:0] store_address_index;
rand bit [`DCACHE_TAG_BITS-1:0] store_address_tag;
rand bit [`LDST_TYPES_LOG-1:0] store_size;
rand bit [`SIZE_DATA-1:0] store_data;

rand bit retry;

rand int delay;

constraint c
{
  load_en dist {0:=8, 1:=2};
  load_size dist {0:=1, 1:= 1, 2:=20, 3:=0};
  load_address_index inside {1,10};
  load_address_tag inside {10,11};
  store_en dist {0:=19, 1:=1};
  store_size dist {0:=1, 1:=1, 2:=20, 3:=0};
  (load_en == 1) -> (store_en == 0);
  retry dist {0:=19, 1:=1};
  //retry == 0;
  delay inside {[2:5]};
}
endclass // core_cache_if

Figure 3.5 core_cache_if class
monitor will not only detect the responses from the DUT, it will also convert them into transactions. Hence, any meaningful exchange on the interface is to be converted into transactions. And only when we have a transaction, we send the monitored packet to other classes. The monitor will also need a copy of the interface since it is the pin-level signals that it is monitoring. The monitor isn't exactly a transactor, because it responds to changes in signal-level information rather than on input transactions. But its structure is similar. It has a run task which spawns threads to monitor the two interfaces and to send the detected input and output transactions to the scoreboard. A snippet of the monitor run task is shown in Figure 3.6.

### Figure 3.6 Monitor run() task

```verilog
// Figure 3.6 Monitor run() task

// if defined MONITOR PRINTS
// $display("Inside monitor run");

fork
  run_cache_in();
  run_send_input_packet();
  run_c2c_out();
  run_c2m_out();
join
endtask
```

### 3.7 Generating ideal responses. Scoreboard:

Scoreboard is the class that generates ideal responses for the inputs applied to the DUT. Since the scoreboard is a transaction-level class, it doesn’t need to generate cycle-accurate responses. This makes the scoreboard very much like a procedural code rather than a RTL-like behavioral code. Thus the scoreboard is much easier to develop, debug and change as compared to the DUT.

Scoreboard is a transactor. In the run task, it receives input transactions from the monitor. It processes the input and generates two set of transactions: the upper-hierarchy/core interface output transaction and the lower-hierarchy/memory interface output transaction. It sends these transactions to the checker through two different mailboxes.
While the scoreboard need not be cycle-accurate, it should be cycle aware in some sense. For example, if an input stimulus generates two different outputs separated by a few cycles. We need to make sure that these two outputs are generated as two different transactions and any output transactions between these two are sent in order. To ensure this, the monitor sends a input transaction every rising-edge of the clock, even if we don’t have any valid input to the scoreboard. It can be thought of that the clock-edge itself is an input to the scoreboard.

**3.8 Comparing the response. Checker:**

The checker class compares the output transactions from the monitor to the ideal output transactions generated by the scoreboard. Since both interfaces have different mailboxes, the checker has four mailboxes in total.

The class has six different threads. Four threads (one for each mailbox) to wait on mailboxes, one thread to compare upper-hierarchy outputs and one thread to compare the lower-hierarchy outputs. We could have manage all the functionality in fewer threads, but having different threads for each functionality makes things much simpler.

The four threads that wait on a mailbox pick the transaction from the mailbox and push it into a dedicated queue for each of the four transactions. They notify the checker thread that a new transaction has arrived. The checker threads check if we have at least one thread each from monitor and scoreboard. If we do, it pops the first transactions from both queues and compares them. It flags an error when it sees a mismatch. We can decide how to handle the error, we could stop the simulation and dump the two mismatching transactions or we could just dump the error transactions and continue with verification.

**3.9 Gathering coverage information. Coverage:**

The coverage class is what takes care of collecting coverage. The monitor sends the input transactions to the coverage class too. In the class we have defined covergroups and coverpoints. The coverage is a transactor, in the sense that has a single thread waiting on communication from the monitor. But, it doesn’t produce an output.
3.10. TESTBENCH CLASS:

In this thread, the class samples all its covergroups to gather coverage on the input transaction. The coverage is also a transaction-level class, it isn't aware of the cycle level behavior of the inputs. Therefore, for measuring coverage on the delays, specific delays need to be measured in the monitor class and sent to the coverage.

3.10 Testbench class:

The testbench class is where all the components of the testbench fit together. We've discussed that every class has its own run() task which is a forever running thread. This consistent structure helps in developing the testbench class.

In its constructor, the testbench class instantiates all the classes that we discussed: driver, monitor, scoreboard, checker and coverage. It sends the necessary arguments to each class's constructor. It has a run() task of its own where it spawns threads for all the component classes. The run task for the testbench is shown in Figure 3.7.

The tester shown in the snippet (Figure 3.7) can be related to any test that we want to run. The specific test will extend the tester module. And it will be instantiated in the testbench class. Hence to run a different test, we replace the tester in the testbench by a different tester.

```vhdl
    task run(input int n);
    `ifdef TESTBENCH_PRINTS
    $display("Inside DCache_testbench run");
    `endif
    fork
      tester_1.run(n);
      driver_1.run();
      scoreboard_1.run();
      monitor_1.run();
      checker_1.run();
      coverage_1.run();
    join
    endtask // run
```

**Figure 3.7** Testbench run() task
3.11 Putting it all together. top module:

Once we've created out testbench, we need a hierarchy to instantiate it and to connect it to the DUT. This is done in a top-level module, which in our case is called the top module. Unlike our testbench components, it is a Verilog module rather than a class. The top module does the following things:

1. Generates the clock for the DUT
2. Instantiates the SystemVerilog interface. It connects the generated clock to the interface
3. Instantiates the DUT. Connects the DUT inputs and outputs to the corresponding interface signals.
4. In an initial block instantiates the testbench class. We send the interface as an argument to the testbench constructor for components that need it.
5. In the same initial block starts the testbench by calling the run() task of the testbench. The argument n to the task is how many packets do we want to generate for the test. This argument is passed to the tester's run() task.

3.12 Comments:

Now that we have the structure of the OOP-testbench, we can make a few observations:

- The testbench is in itself quite modular. Components handle very specific tasks. Tasks that are related but can be changed are separated into different modules. For example, new tests can be added by changing the tester class, without changing any other class.
- Layered structure and a transactor skeleton works well as we can instantiate a single class and just call the run task in the top hierarchy for all the classes to start executing.
- The DUT interface is passed to two different classes: driver and monitor. No other class uses it. The interface has to be passed through the testbench class, not because the testbench class itself needs it, but because component modules may need it.
3.12. COMMENTS:

- Changes to the DUT signals will need changes in interface, driver and monitor.

- The monitor needs to send the input transaction to both scoreboard and coverage classes. It needs to have different mailboxes for both. In future if we divide the coverage class into further classes or any other class is introduced which needs the input transaction, we will need to change the monitor to add another mailbox to it.

- To run different tests on the testbench, we need to inherit the tester class and override the get_transaction() function of the class to generate new stimulus. This new class will then be instantiated in the testbench. So, we need to change our testbench class and instantiate a new testbench class in the top module for every new test. Even for tests for which the corresponding tester classes were already created, we need to change the testbench code, re-compile it and then use it. This makes switching between tests tedious.
We'll focus on the structure and the development of the UVM-testbench in this chapter. We describe how the baseline OOP-testbench was modified to use the UVM-features. UVM is a feature rich methodology which can be used for designs much more complicated than the one we are concentrating on. Unlike the OOP-testbench where we can have one test in a compile, UVM allows us to have a suite of tests ready to run without any need to recompile. This is why we call it a verification suite instead of a testbench.

We won't be going into the details of all the features that UVM offers. We want to focus on the features we used in making our testbench better. First, in section 4.1 we will delve into the UVM features that we have used. In section 4.2 we will discuss the changes that we made to the OOP-testbench using these features.
4.1 UVM features

In this section, we'll be describing UVM features and base classes that we've used. While the description of each class doesn't go into the details of all the component methods and data, we'll concentrate on the parts that are critical to understanding the changes we made. The UVM Cookbook[2] is an excellent resource to understand all the implementation details of the base-classes.

4.1.1 uvm_component and uvm_object

A UVM verification environment has two types of objects: the objects that form the structure and functionality of the testbench and objects that carry data for the testbench. UVM divides all its classes into these two broad types.

- Every class which is designed to act as a structural and functional aspect of the testbench is inherited from a uvm_component base class, e.g., driver, monitor, scoreboard.

- Every class that is designed to carry data is inherited from a uvm_object base class, e.g., transactions, coverage data, monitor input packets, etc.

The routines and data inside these base classes differ according to the purpose of each base-class. All the other classes in UVM are inherited from either of these two classes.

4.1.2 UVM factory

The UVM Factory Pattern is a design pattern[3] that creates objects for us. Every class in the UVM hierarchy is registered with the UVM Factory. The Factory can create objects that are registered with it. It is like a large case block which selects which object to create depending on the arguments that are passed to it. The Factory also correctly typecasts the object created according to its LHS class type. It is up to the user of the Factory to ensure that such typecasting will be valid.

The purpose of using the Factory is to change the type of object to be created not by changing the code, but by changing arguments to the Factory during simulation. In many
situations, we want to change objects in the testbench based on many factors, e.g., the test that is being run currently. In the testbench class in the OOP-testbench Section 3.10, we instantiate and run different tester objects depending on the type of test we want to run. Instead of hard-coding the type of tester object we create in the testbench[3], if we create the object using the UVM Factory, we can change the type of test by providing different arguments during the simulation (Figure 4.2). We don’t need to change the testbench class or recompile the whole testbench. In a complicated testbench, there will be multiple instances where we would want to change the type of object to instantiate depending on many factors. The UVM Factory will be useful in every such instance.

We use macros provided by UVM to register classes with the Factory. uvm_component and uvm_object are registered using different macros. Classes extending uvm_component register themselves using the macro: uvm_component_utils and classes extending uvm_object register using a different macro: uvm_object_utils. Figure 4.3 is an example of a class registering with the factory.
4.1. UVM FEATURES

CHAPTER 4. UVM-BASED VERIFICATION SUITE

Class instances are instantiated using a different `create()` function defined by UVM instead of a constructor call so that the instantiation goes through the Factory. Figure 4.4 is an example of the use of create() function for the class we registered.

The create() function for uvm_objects takes one argument

- type **string**: name of the object to uniquely identify it.

For uvm_component create() takes two arguments

- type **string**: name of the component to uniquely identify it.

- type **uvm_component**: parent component of this component. When a class instantiates an object it passes its own pointer, i.e., the `this` pointer as the parent. This helps UVM form a graph of the hierarchy of the testbench.

### 4.1.3 UVM configuration database:

Some objects or variables values (like the SystemVerilog interface) are needed by multiple components. Instead of passing these objects or values throughout the testbench, UVM uses a configuration database to store these. Components that need the value retrieve them from the database. Figure 4.5 is an example of the an BFM object being stored in the uvm_config_db in the top module. We'll see more about the BFM in section 4.2.1. The object can be retrieved later using the get() function of uvm_config_db by the driver (Figure 4.6).

The database is parametrized by the type of object being stored. In our case it is a "virtual dcache_interface". The set() function is used to store the value. The get() function is used to
4.1. UVM FEATURES

4.1. UVM FEATURES CHAPTER 4. UVM-BASED VERIFICATION SUITE

Figure 4.5 Storing in uvm_config_db

```verbatim
initial
begin
  uvm_config_db #(virtual dcache_interface) ::set(null,"*","dcache_if",dcache_if);
  run_test();
end
```

Figure 4.6 Retrieving from uvm_config_db

```verbatim
if(!uvm_config_db #(virtual dcache_interface)::get(null, "*","dcache_if", dcache_if))
  uvm_fatal("DRIVER", "Failed to get interface");
```

retrieve it. The first two arguments are the storage location for the object handle. We want the object to be available throughout the entire testbench, so our arguments are "null" and "*". The third argument is the name of the data to be stored and the fourth is the data itself. The name and the parameter arguments are needed when retrieving the data from the uvm_config_db.

4.1.4 UVM phases

The UVM factory helps in constructing the testbench by instantiating all the testbench objects for us. The user needs to specify the hierarchy by instantiating objects inside the components using the create() instead of new() functions. UVM needs the testbench creation and simulation to run in a particular order. This is enforced by certain functions and tasks in the uvm_component class that the child classes should override.

The UVM testbench simulation cycle is divided into what are called phases. We’ll take a brief look at some of the UVM phases relevant to our testbench.

- **build phase**: All the objects in the structure are instantiated in this phase. The simulation starts with the UVM root node component being constructed. After this, the build_phase() methods of all the components are called in top-down order of hierarchy. So, first the top-most class is instantiated, then its build_phase function is called. If a component needs to instantiate objects, it should do so in the build_phase() method. The build_phase functions for all these instantiated objects is then called. And the process goes on till all the objects have been instantiated. Note that we're
talking about instantiating testbench components. Hence, uvm_objects relevant to a uvm_component's functionality need not be instantiated in the build_phase.

- **connect phase**: The various connections between components are made in this phase. This phase is called after the build phase execution is complete, so UVM assumes that all components have been instantiated. In this phase, connect_phase() function of all components are called in the bottom up hierarchy order. Each component makes the connections for its internal objects in its connect_phase() method.

- **end of elaboration phase**: This phase is executed after the connect phase. Adjustments to the testbench to be made after the connect phase, if any, are made in this phase. This is done by calling end_of_elaboration_phase() function for all uvm_components in bottom up order.

- **run phase**: This part of the simulation is the key to the execution of each test. The UVM simulation engine starts this phase for all components simultaneously by calling their run_phase() tasks. Like in SystemVerilog "fork-join" construct, the execution sequence for the run_phase tasks of all the components is non-deterministic. So the synchronization between the run phases of all components should be taken care of by the testbench designer. The stimulus generation and application, checking of signals and coverage gathering happens in this phase.

- **report phase**: When the run phase execution concludes, report_phase() functions for all the components is called. This is used for the purpose of reporting vital stats related to the executed tests.

UVM has many other simulation phases which we aren't focusing on now. These other phases too are handled by specific methods in each component class. The uvm_component definition comes with these phase methods. The components wanting to use any phase must override the corresponding phase method.

Of the phases we discussed, except the run_phase all other phases are functions. The run_phase is a task since it needs simulation time to generate stimulus and check results. Figure 4.7 is an example of implementation of the build_phase.
Not all components may need all the phases. For example, certain components may not have any components within them and may not need to override the build_phase method. Certain components may not participate directly in the stimulus generation or monitoring responses part of the testbench, they might be in place to define structure rather than function. Such components won't need to override the run_phase task.

### 4.1.5 Raising objections

Since, the UVM itself instantiates and runs the phases for all the component classes, it cannot decide on itself when to terminate the run_phase. So, it follows a simple rule: it terminates a phase if not explicitly told to wait. Usually for a testbench, the run_phase is where most of the simulation occurs. It is also the phase which can consume simulation time. In the OOP-testbench, the forever loops for all the run() tasks ensured that we won't stop the simulation. We explicitly stopped the simulation when we were done generating stimulus. We do the opposite in the UVM testbench: we explicitly tell the UVM simulation engine to wait till we are done.

This is done through objections. Each phase related function/task has an argument called phase which is of type uvm_phase. This argument is used by the UVM simulation engine to communicate with the component objects. When a component wants the simulation to stay in the current phase, it raises an objection using the phase's raise objection...
4.1. UVM FEATURES

4.1.6 Observer design pattern[3]

In Chapter 3 comments section 3.12, we observed that in some cases of the OOP-testbench, we needed multiple mailboxes to send the same data to different modules (Figure 4.9). While this works, this limits the reusability of the modules. If in future we need to add more modules that need the same data, we’ll need to add more mailboxes sending the same data. We’ll also have to add connections between the new module and the generating module. The connections of a monitor to scoreboard, monitor to checker and monitor to coverage are very typical of every testbench. And the multiple mailboxes problem will arise in every case. UVM has the observer design pattern to simplify cases like this. It is useful in every case where there is a single generator and many consumers (or what UVM calls observers).

In our case, the monitor would be the generator and rest of the modules will be observers.

```verilog
task run_phase(uvm_phase phase);
    reset_sequence reset_seq;
    line_load_sequence load_seq;

    reset_seq = reset_sequence::type_id::create("reset_seq");
    load_seq = line_load_sequence::type_id::create("load_seq");

    phase.raise_objective(this);

    reset_seq.start(sequence_h);
    load_seq.start(sequence_h);

    phase.drop_objective(this);
endtask // run_phase
```

Figure 4.8 Raising Objections

method, letting UVM know that it needs to stay in this phase. This acts like a flag which UVM checks before moving on to the next phase. If a single objection is raised, UVM waits in that phase. Once the component is done, it can drop the objection using the drop_objective method in the phase. When all objections are dropped, UVM moves on to the next phase.

Figure 4.8 is a code snippet showing a test raising an objection before generating stimulus and dropping an objection after it is done. raise_objective and drop_objective have one argument which is the pointer to the component raising the objection.
UVM implements observer design pattern using the uvm_analysis_port and analysis_export constructs (Figure 4.10). These constructs, like mailboxes, take parameters of the type of data we want to send. Specifically, they can only communicate with each other through classes of type uvm_transaction.

4.1.6.1 uvm_transaction:

The uvm_transaction is inherited from the uvm_object and acts as a base class to all transaction level objects used in the testbench. For example, data passed between the monitor and the scoreboard should be of type uvm_transaction. It inherits all the properties of uvm_objects and provides timestamp properties, transaction recording properties.

4.1.6.2 uvm_analysis_port:

The uvm_analysis_port is inherited from uvm_component. It can connect and communicate to another UVM construct called analysis_export through objects of type uvm_transaction. A single uvm_analysis_port can connect to multiple analysis_exports. It is a parametrized class, the parameter being type of uvm_transaction that can be sent using the port. uvm_analysis_port has a write() function. Argument to write() function is the data to be sent to the observers. The parent class instantiating the uvm_analysis_port calls the write() function and sends the data object as an argument. On a call to the write() function,
the `uvm_analysis_port` calls the `write()` functions of all `analysis_exports` connected to it and sends them the data passed to these `write()` functions.

For our generator-observer design pattern, the generator instantiates a `uvm_analysis_port` in it. All the classes that wish to be an observer for the generator, connect to this `uvm_analysis_port`. We'll see in further sections how to connect to `uvm_analysis_ports`.

### 4.1.6.3 `uvm_subscriber`:

`uvm_subscriber` is extended from the `uvm_component`. Along with the `uvm_component` routines, it provides an `analysis_export` to connect to a `uvm_analysis_port` and it also provides a `write()` function. It is a parametrized class, the parameter being of the type `uvm_transaction` the `analysis_export` can receive.

Observers extend the `uvm_subscriber` and override the `write()` function to handle the incoming data from the generator. The observer's `analysis_export` is connected to the `uvm_analysis_port` of the generator. An `analysis_export` can be connected to only one `uvm_analysis_port`.
When the generator sends a data object to the write() function of its uvm_analysis_port, the write() functions of all analysis_exports connected to it are called. Which means, the observer's own write() function will be called with the data object being passed as an argument. Thus the write() function is where the observer needs to handle the incoming data.

Point to be noted here is that the write() is a function and hence cannot consume simulation time or have any synchronization elements. The communication between uvm_analysis_port and uvm_subscribers, therefore, cannot be inter-thread communication. All the observer write() functions are called in the same thread as that of the generator's write() function. If the subscriber has its own thread, the communication between its write() function and its thread will have to be handled by the subscriber itself.

Since the uvm_subscriber has to be extended, a component can extend uvm_subscriber if it needs to observe only one uvm_analysis_port. If a class needs to observe multiple analysis ports, it can use another construct called the uvm_tlm_analysis_fifo.

4.1.6.4 uvm_tlm_analysis_fifo:

For components that need to observe multiple uvm_analysis_ports, UVM provides the uvm_tlm_analysis_fifo construct. This is also a parameterized class and can be instantiated inside a component, like a mailbox, instead of extending a base class like uvm_subscriber. It provides a analysis_export to connect to a uvm_analysis_port.

We don't need a write() function override for the uvm_tlm_analysis_fifo. Communication between the uvm_analysis_port and uvm_tlm_analysis_fifo happens as inter-thread communication. On calling the uvm_analysis_port's write() function, the object to be communicated is written to a FIFO associated with the uvm_tlm_analysis_fifo object. The uvm_tlm_analysis_fifo has a blocking get() function which can be used to get packets from the FIFO.

Since uvm_tlm_analysis_fifo are to be instantiated within components rather than extending them, we can have multiple analysis FIFOs in each component. Each analysis fifo can still connect to one and only one uvm_analysis_port.
4.1.7 **uvm_sequence_item:**

UVM introduces a new class type uvm_sequence_item. This base class is developed to be used for stimulus transactions between the test and the driver. uvm_sequence_item extends the uvm_transaction class. Along with all the uvm_transaction features it provides methods to help in generating sequences like setting sequence info which can help in generating future sequences.

In UVM test stimulus packets are called sequence items instead of transactions. This terminology helps in differentiating between types of data being used in the testbench.

4.1.8 **uvm_driver:**

UVM defines a new class called uvm_driver which is inherited from uvm_component to implement drivers efficiently. The uvm_driver is a parametrized class, with the parameter being a type of uvm_sequence_item.

The uvm_driver has a seq_item_port (sequence item port) which is designed to receive sequence items. Like a mailbox, the seq_item_port can receive objects. The advantage of seq_item_port over a mailbox is that the receiver can send packets back to the sender once it is done processing the packet. The driver can give feedback to the test after processing a sequence item with this feature. This feedback can be the DUT output for the input we applied. This is very useful for stimulus generation since we can change stimulus depending on the type of feedback we are receiving. Even if we don't use the feedback in an intelligent way, it is useful for the test to get an indication when the previous transaction was done processing and was sent to the DUT.

The seq_item_port connects to an object of seq_item_export which is used to send the sequence items. The seq_item_port has the get_next_item() and the item_done() methods to communicate with the seq_item_export. Argument to get_next_item() is of type uvm_sequence_item. The item_done() method doesn't have any arguments. It is an indication to the sequencer that the processing is complete. The uvm_driver receives the sequence items using get_next_item(), and can send it back using other methods like put_response(), which we aren't using for our testbench.
4.1.9 uvm_sequencer:

In the OOP testbench, our tester class does two tasks:

1. generate stimulus.
2. send the stimulus to the driver through a mailbox.

The uvm_sequencer is created in UVM to separate these two tasks. uvm_sequencer takes a parameterized definition. The parameter should be of type uvm_sequence_item. The uvm_sequencer has a seq_item_export which can connect to the driver's seq_item_port. A uvm_sequencer is usually used by tests to send transactions to the driver. Separating generation from communication may not seem like a big change for the tests. This change implies that we can hardcode the connection to the driver without needing to hardcode stimulus generation. Thus the connection to the driver is set in place, while we can play around with what can be connected to it. This effectively makes the stimulus-generation independent of the structure. This also means that we can generate stimulus from multiple places at the same time and use the same connection to chose between different stimuli. This gives us a fertile ground to reuse existing tests. We'll explain how to exploit these advantages when we cover the UVM tests (and a related concept: sequences) in chapter 5.

Now that we've spent time looking at stand-alone features of the UVM. We'll look at the steps that we followed to convert the OOP-testbench to UVM-testbench.

4.2 Creating UVM testbench

4.2.1 Step 1: Changing Interface to BFM[3]

The first step in converting the OOP-testbench to UVM is to create a construct called Bus Functional Model (BFM). The BFM is a improved version of the interface that we used in the OOP-testbench. Like the interface, it is instantiated in the top module of the testbench. BFM isn't related to the UVM features and can be used in a testbench not compliant to UVM. But it helps us in using the UVM features better.

The interface in OOP-testbench (section 3.1.) solved the problem of managing complexity of the various input/output signals of the DUT. But as we mentioned in the chapter
ending comments, the driver and monitor classes will have to change when we change the
interface signals. The SystemVerilog interfaces can also have functionality in them, like
tasks, functions and other functional blocks. The BFM is the interface (with the bundles
of input/output signals) along with all associated functionality related to the signals. For
example, in addition to the signals, it has a task which takes in arguments and drives the
core-input signals accordingly. This functionality was previously handled by the driver. If
we shift all such functionality in the interface, this keeps all the signals and their related
functionality in one file.

Along with the functional blocks, we also add internal DUT signals to the interface.
These signals help in gathering coverage information from the DUT. For example, we can
get information as to whether an input load address is present in both the cache line and the
store buffer.

The BFM for our testbench is in the dcache_interface.sv file and we've added the follow-
ing functional blocks:

- send_input() task: This task sends inputs to the DUT from both the core/upper-
hierarchy and the memory/lower-hierarchy interface.

- always@(posedge clk) blocks: Handles the memory/lower-hierarchy interface of the
DUT and model the memory functionality.

- The BFM has a copy of the monitor instance created for the testbench. It has always@
blocks sample the inputs to the DUT. Sends the monitored inputs to the monitor
class using the write_to_input method in the monitor.

- always@ blocks: Samples all the outputs on the core-cache interface. Sends the out-
puts to the monitor using the write_to_core_output method in the monitor.

- always@ blocks: Samples all the outputs on the cache-memory interface. Sends the
outputs to the monitor using the write_to_memory_output method of the monitor.

- always@ blocks: Samples all the coverage related signals and send them using the
write_to_coverage method of the monitor.

Our first step in changing the OOP-testbench to UVM is to develop a BFM.
4.2.2 Step 2: Change all data packets to uvm_transactions:

We have many different classes to carry data between the testbench components. Now that we're constructing a UVM compliant testbench, we'll need to make sure that all our data objects are of the type uvm_transaction. This is going to be our next step. We'll change all our data classes to uvm_transaction. Exception to this is the core_cache_if class which is used for generating test stimulus and will be modified to type uvm_sequence_item. We will see this in a latter section. The classes that we are modifying are the ones that carry data between monitor to scoreboard, monitor to coverage, monitor to checker and scoreboard to checker. Steps to convert classes to uvm_transaction:

1. Extend class definition from uvm_transaction.
2. Register it with Factory using uvm_object_utils.
3. Override the constructor.

We change the cache_in, cache_core_out, cache_memory_out coverage_trans classes to uvm_transaction.

4.2.3 Step 3: Modify Monitor to uvm_component

Let's have a look at how the monitor should change in the UVM-compliant testbench. The monitor in UVM-testbench will first extend the uvm_component and register itself with the factory (Figure 4.11).

It won't have a run_phase task since all the monitor operation is now being handled in the BFM. In this testbench, monitor class is the structural element defining where the monitoring will happen in the testbench, but doesn't handle any functional part of monitoring. It will have uvm_analysis_ports to send all the monitored packets (Figure 4.12). It will need a build_phase method to instantiate these ports (Figure 4.13).
In OOP-testbench, we were getting the interface from the constructor. We cannot pass any other arguments in the constructor when extending uvm_component since the arguments are strictly defined by the uvm_component class which we have to override. We instead use the uvm_config_db to get the BFM in the build_phase method. Once we get the BFM, we need to connect the BFM’s copy of monitor to this component (Figure 4.13).

While, the connection should ideally be done in the connect_phase for all components, BFM isn’t a testbench component in the proper sense, it is instantiated in the top module so we can be sure that it is instantiated before the build_phase is called.

We’ll need to define functions that the BFM can use to communicate the signals monitored with the monitor. We have four different functions for this purpose. These functions are called from BFM to pass signals to the monitor. Inside these functions we construct the uvm_transaction objects and send them using write() functions for each of the analysis ports.

The steps for changing the monitor are as follows:

1. Extend the uvm_component to create the monitor and register it with Factory. Override the constructor of the uvm_component.
4.2. CREATING UVM TESTBENCH  

CHAPTER 4. UVM-BASED VERIFICATION SUITE

2. Use uvm_analysis_ports instead of mailboxes.

3. Override the build_phase method. Get the BFM from uvm_config_db. Connect the monitor to BFM’s copy of the monitor.

4. Create functions that the BFM can use to send each of the four packets. Instantiate uvm_transacation packets in these functions using the input values to the function and call write() functions for the corresponding uvm_analysis_ports to send these packets.

4.2.4 Step 4: Modify the Checker to uvm_component:

The checker will be an observer to the monitor’s analysis ports for getting the outputs on the two interfaces. And it also needs the scoreboard’s ideal output. Since, the checker needs to be an observer for multiple analysis ports, we use uvm_tlm_analysis_fifo in the checker. The mailboxes in the OOP-testbench are replaced by uvm_tlm_analysis_fifos (Figure 4.15).

The checker will have a build_phase (Figure 4.16) to instantiate the analysis FIFOs, and a run_phase (Figure 4.17) to spawn threads to wait on the FIFO get() methods and few other
threads to check the results.

The threads used in the OOP-testbench will stay largely the same in the checker for UVM. Steps to change checker:

1. Extend uvm_component and register with the Factory. Override the constructor.
2. Use uvm_tlm_analysis_fifo instead of mailboxes. Parametrize them corresponding to their data classes.
3. Override the build_phase to instantiate the uvm_tlm_analysis_fifos
4. Change run() task to override the run_phase. The spawned threads remain largely the same. They wait on analysis FIFOs instead of mailboxes.

### 4.2.5 Step 5: Modify the Scoreboard:

The scoreboard needs the cache input from the monitor. So it needs to be an observer for only one analysis port. So it extends the uvm_subscriber.
It needs to override the uvm_subscriber’s write() function to get data from the monitor’s analysis port. In the write() function, we copy the packet we get from the monitor and push it to a local queue. Since the write() function will be in a different thread, we use events to synchronize the scoreboard thread with the write() function. Scoreboard also needs to communicate the ideal outputs to the checker, for which we use a uvm_analysis_port for each of the two output interfaces, instead of mailboxes used in OOP-testbench. Scoreboard will need to override the build_phase to instantiate the analysis ports.

The run() task in the OOP-testbench scoreboard changes to the run_phase task. The functionality of the scoreboard remains largely the same for the two test-benches. Steps to change scoreboard:


2. Use uvm_analysis_ports instead of mailboxes to send ideal outputs.

3. Implement write() function. Copy the transaction received in the function into a local queue and maintain synchronization with run_phase using events.

4. Override build_phase to instantiate the uvm_analysis_ports.
5. Modify the run() task to override run_phase task. Wait on the synchronization event from the write() function instead of mailbox. Get the data from the local queue instead of the mailbox.

4.2.6 Step 6: Modify the Coverage:

The coverage class will also extend the uvm_subscriber since it gets its input data from the monitor’s analysis port.

For the UVM testbench, we have made one significant change in the way we measure coverage. Initially, the cache input packet that monitor sent to the coverage was the same that was being sent to the scoreboard. But, the coverage needs a lot more information than the inputs. It also needs the internals of the DUT to measure coverage for various responses to the DUT. It also needs the outputs to measure coverage against all types of possible outputs. So, the scoreboard and coverage each need different data from the monitor. We decided to separate the coverage data from the scoreboard data completely. Now it is handled by a different analysis port in the monitor.

Our argument for using analysis ports was that we don’t need different mailboxes for the same data, like scoreboard and coverage. While adding a different analysis ports for scoreboard and coverage negates our argument for using analysis ports in the first place, it still remains true that using analysis ports is much more reusable than using mailboxes. It gives us the ability to add observers in future without changing the classes. And since the data being sent to both scoreboard and coverage is different, it makes sense to have...
different analysis ports for both.

While adding many signals to the coverage class, we’ve also added new covergroups to measure coverage for many different scenarios. Chapter 6. goes into the details of coverage metric. Steps to change coverage class:


2. Implement write() function. Copy data to a local queue on receiving the input coverage data and use events to synchronize with the run_phase.

3. Modify run() task to override run_phase task. Use synchronization event and local queue for synchronization and data instead of the mailbox.

4.2.7 Step 7: Modify core_cache_if:

In OOP-testbench, we have a class called core_cache_if which was used to generate the test stimulus and send it to the driver. In UVM, because uvm_sequencer (section 4.1.9) and uvm_driver (section 4.1.8) take in parameters in the form of uvm_sequence_item, we need to change our core_cache_if to make it of uvm_sequence_item type. We change it to extend uvm_sequence_item and use uvm_object_utils to register it with factory (Figure 4.22). Rest of the contents of the class stay the same. Steps to change core_cache_if:
4.2. CREATING UVM TESTBENCH  CHAPTER 4. UVM-BASED VERIFICATION SUITE

4.2.8 Step 8: Modify the Driver:

The driver in our UVM testbench extends the uvm_driver class with the core_cache_if as the parameter (Figure 4.23). A major change to the driver is that since we are using a BFM instead of an interface, the driver doesn’t need to apply the stimulus to the signals. Like the monitor, it plays a structural role. It receives communication from the sequencer about the stimulus to be sent and sends it to the BFM to apply. The BFM has the send_input method to get data from the driver. Second major change to the driver is the way we receive transactions. Since extending the uvm_driver gave us access to seq_item_port, we’ll receive transactions using this port. The run() task will change to run_phase (Figure 4.24). We’ll use the get_next_item and item_done methods of the seq_item_port to communicate with the sequencer and the send_input method to communicate with the BFM. Like the monitor, the driver gets the BFM from the uvm_config_db using the get() function. It does this in the build_phase.

Steps to modify driver:

1. Extend uvm_driver with parameter core_cache_if. Register with Factory. Override the constructor.

2. Override build_phase. Get the BFM from uvm_config_db in build_phase.

3. Override run_phase. Get data using seq_item_port’s get_next_item() method. Send
task run_phase(uvm_phase phase);
    core_cache_if transaction;
    forever
        begin
            seq_item_port.get_next_item(transaction);

            `uvm_info("DRIVER", "Transaction from sequencer received", UVM_HIGH);

            dcache_if.send_input(transaction.reset,
                transaction.load_en,
                transaction.load_address_offset,
                transaction.load_address_index,
                transaction.load_address_tag,
                transaction.load_size,
                transaction.store_en,
                transaction.store_address_offset,
                transaction.store_address_index,
                transaction.store_address_tag,
                transaction.store_size,
                transaction.store_data,
                transaction.retry,
                transaction.delay_between_commands,
                transaction.delay_for_load_reply,
                transaction.mem_load_data,
                transaction.delay_for_store_reply,
                transaction.invalidate_valid,
                transaction.invalidate_address_index,
                transaction.invalidate_address_tag,
                transaction.invalidate_delay,
                transaction.scratch_mode_enable,
                transaction.scratch_wr_addr,
                transaction.scratch_wr_en,
                transaction.scratch_wr_data);

            seq_item_port.item_done();
        end // forever begin
    endtask // run_phase

Figure 4.24 Driver run_phase()
4.2. CREATING UVM TESTBENCH

4.2.9 Step 9: Create dcache_agent:

We’ve now modified the monitor, scoreboard, checker, driver to make it UVM compliant. In the OOP-testbench, we get all of them together in the testbench class and instantiate them there. We make the connections between them in the same class.

We can collect these classes in a much better way so that we get more reusability out of them. All these classes will be used to verify one cache. If in the future we need to verify a larger design with the cache being a part of the design or a design with multiple caches, we should be able to reuse the classes that we developed. We can do that by instantiating these classes in one component in the hierarchy called an agent. This agent is the class in hierarchy which carries all the necessary components to verify a single cache and can be instantiated multiple times to verify multiple caches.

The external inputs to the agent class will be the BFM and the stimulus. Since the BFM is retrieved through the uvm_config_db, we don’t need to explicitly get this as an input. The stimulus will be received through a sequencer. Hence in the agent, we will need to instantiate a sequencer which can be connected to the outside world.

the data to the BFM to apply to the DUT. Call the seq_item_port’s item_done() method to indicate that processing of this transaction was done.
Figure 4.26 Agent build_phase()

```verbatim
function void build_phase(uvm_phase phase);
    sequencer_h = new("sequencer_h", this);
    driver_h = driver::type_id::create("driver_h",this);
    monitor_h = monitor::type_id::create("monitor_h",this);
    scoreboard_h = scoreboard::type_id::create("scoreboard_h",this);
    checker_h = cache_checker::type_id::create("checker_h",this);
    coverage_h = dcache_coverage::type_id::create("coverage_h",this);
endfunction // build_phase
```

Figure 4.27 Agent run_phase()

```verbatim
function void connect_phase(uvm_phase phase);
    driver_h.seq_item_port.connect(sequencer_h.seq_item_export);

    monitor_h.cache_input_port.connect(scoreboard_h.analysis_export);
    monitor_h.cache_core_out_port.connect(checker_h.monitor_core_out_fifo.analysis_export);
    monitor_h.cache_memory_out_port.connect(checker_h.monitor_memory_out_fifo.analysis_export);
    monitor_h.coverage_out_port.connect(coverage_h.analysis_export);

    scoreboard_h.core_out_port.connect(checker_h.scoreboard_core_out_fifo.analysis_export);
    scoreboard_h.memory_out_port.connect(checker_h.scoreboard_memory_out_fifo.analysis_export);
endfunction // connect_phase
```

We define the agent class by extending the class uvm_agent. This is a child class of uvm_component. We are instantiating all the classes in agent, so we'll need to override build_phase (Figure 4.26). And since we've to make all connections too, we'll override the connect_phase function (Figure 4.27). We aren't running any functionality in this class, we don't have to override run_phase. In addition to the phase methods, uvm_agent has a get_is_active() method which is used to figuratively switch the agent on and off. This is particularly helpful if we have multiple caches in our DUT and we would like to chose which ones to verify. Since, our testbench has only one cache as the DUT, we don't use the get_is_active() method. We've instantiated all the component classes and made all the connections. We've connected the sequencer to the driver too. A class using dcache_agent can get access to the sequencer by using the dot notation: agent.sequencer_h. Steps to create agent:

1. Extend uvm_agent to create a new class dcache_agent. Register it with Factory. Override the constructor.
2. Override build_phase. Instantiate all components: driver, monitor, scoreboard, coverage, checker and a sequencer. The parameter to the sequencer will be core_cache_if.

3. Override connect_phase. Connect uvm_analysis_port to analysis_export of its observers. Connect driver’s seq_item_port to the sequencer’s seq_item_export.

This completes our structure for the verification of a cache. Now we need to think about how to instantiate the agent and run specific tests on it.

### 4.2.10 Step 10: Create an env class:

The dcache_agent defined in the above section can be instantiated in a test which can then be called. If there are more than one caches in the DUT and if we would like for them to interact in some way, we’ll need an intermediate stage before the test itself to do that. We call this stage the environment.

We extend a class called uvm_env which is a child class of uvm_component. The only difference between uvm_env and uvm_component is that uvm_env contains only structure specific phases like build_phase, connect_phase. We instantiate the dcache_agent in the
4.2. CREATING UVM TESTBENCH  CHAPTER 4. UVM-BASED VERIFICATION SUITE

Figure 4.29 Env

build_phase. We define a sequencer here too, but instead of instantiating a new one, we connect it to the one in agent in connect_phase. Steps to create env class:

1. Create the class by extending uvm_env class. Register it with the Factory. Override the constructor.
2. Override build_phase method. Instantiate agent and sequencer in build_phase.
3. Override connect_phase method. Connect the sequencer in env to the sequencer in agent.

Now we have a env component with the sequencer connected to the driver. We are ready for our base_test class.

4.2.11 Step 11: Create base_test class:

Now that we have our structure in place, we need to develop tests to generate stimulus. UVM needs the tests to be of type uvm_test. uvm_test is inherited from uvm_component. We
need the tests to be of a different type than other components in the testbench because the tests are the components from where hierarchy building will begin at the start of simulation. And since we don’t explicitly instantiate tests anywhere, UVM simulation engine needs a way to differentiate between the test and other components.

We create a class called base_test which extends uvm_test. Since it is a uvm_component, we register it with the factory using uvm_component_utils. We instantiate the environment in this class in the build_phase function. We define a sequencer here and connect it to the one in the environment in the end_of_elaboration function. This function corresponds to the end of elaboration phase which is called after the connect phase.

So, we have a class of type base_test, which other tests can extend. Extending the base_test gives any generic test a connection into the driver through the sequencer without the complication of knowing the testbench structure. The sequencer acts like the plug where we can plug different tests. The test is effectively outside the testbench hierarchy and communicates only through the sequencer. This is what we had set out to achieve. We describe the procedure of extending the base_test and developing new tests in Chapter 5.

**4.2.12 Step 12: Create the top module:**

The top module is mostly the same as that for OOP-testbench. We define the clock, instantiate the DUT and the BFM. We don’t instantiate a test specifically. As we’ve discussed, test instantiation will be done by the UVM simulation engine using command line arguments. The only thing needed to connect the test to the DUT is the BFM, which we pass to the uvm_config_db. We also need to call a function called run_test() which is UVM simulation engine’s cue to start executing. We do both these things in an initial block (Figure 4.30).

We need to also include the UVM package to get the definitions of all the UVM base classes that we used. This completes our overall testbench structure. In later chapters, we
discuss how we develop stimulus for this testbench and how we add new tests. We also discuss the coverage metric and the stimulus we defined to meet the metric.

4.3 Comments:

• UVM builds on all the pros of the OOP-testbench and gets rid of most of the cons.

• We have separated structure from functionality in many cases, especially in cases like the driver and monitor where the signals are bound to change in the future. We’ve done the same thing for test stimulus where the stimulus is going to change for each test.

• We’ve made connections between modules easier and reusable by the observer design pattern.

• We have a truly plug-and-play structure for the test stimulus. The base_test has no visibility to the structure of the testbench. The testbench structure has no visibility as to how the tests are generated. We could change either of them without changing the other.

• We mentioned this a few times, but haven't seen this yet: simulating different tests is just a matter of changing the command line arguments for the simulation. This is because of the UVM factory. It was designed so that we can pass arguments to it to instantiate different tests. Even if certain tests need different components, that too can be managed by the UVM Factory.

We’ll see a few other advantages of using UVM in the coming chapters. We'll especially see that adding new tests and reusing old tests is simple.
We've discussed the structure of the UVM-based testbench in the previous chapters. In this chapter we will describe step-by-step process to add a new test to the environment and how to run the test. This chapter is written to be independent of the other chapters in the thesis. The reader may skip chapter 4 if desired, describing the UVM-testbench internals and get the gist of adding a new test by reading this chapter.

What makes this is a verification suite is the ability to keep on adding tests to the existing framework without any changes to either the structure of the testbench-DUT interface or the testbench organization. The tests act as stimulus generators from outside the testbench hierarchy. We'll see how that is achieved and we'll present steps to add new tests to the already existing tests that we have developed.
5.1. **BASE_TEST CLASS:**

We have developed a base_test class in the test suite to act as skeleton to develop all future tests. We’ve discussed this before in section 4.2.11. We’ll have another look at the class from a test stimulus perspective.

The interface which helps us plug-and-play different tests is the "sequencer". This is an example of a well-defined, self-sufficient interface defined by UVM. Important to note here is that the sequencer is part of the env class. We need a way to connect different test classes to the sequencer which is part of the env class. The base_test establishes this connection with the sequencer so that all future tests need not repeat the same code. In short, the base_test class keeps connection to the driver invisible from other tests. base_test extends the uvm_test class. It is a uvm_component and hence uses the uvm_component_utils to register itself. It does the following things:

1. Define the env and the sequencer (Figure 5.1).
2. Instantiate the env in the build_phase (Figure 5.2).
3. Connect the sequencer to the one within the env (Figure 5.3).

The sequencer in the env class is connected to the driver’s seq_item_port. Once the connection between the sequencer in the env has been made to one used in the the base_test, we have ensured that using the sequencer in the base_test, we can affect the driver directly through its seq_item_port.

```c
env env_h;
 uvm_sequencer #[core_cache_if] sequencer_h;
```

**Figure 5.1** base_test: Define env and sequencer

```c
function void build_phase(uvm_phase phase);
  env_h = env::type_id::create("env_h",this);
endfunction // build_phase
```

**Figure 5.2** base_test: build_phase
5.1. **BASE_TEST CLASS:**

**CHAPTER 5. ADDING A TEST**

```verilog
define void end_of_elaboration_phase(uvm_phase phase);
    sequencer_h = env_h.sequencer_h;
endfunction // end_of_elaboration_phase
```

**Figure 5.3** base_test: end_of_elaboration_phase

**Figure 5.4** base_test sequencer connections
5.2 UVM_SEQUENCE_ITEM:

UVM introduces a new class type: uvm_sequence_item for defining stimulus carrying transactions. This helps us differentiate between the stimulus transactions and other transactions. uvm_sequence_item is extended from uvm_transaction hence it uses the uvm_object_utils to register itself with the UVM Factory.

Testbench components like uvm_driver, uvm_sequencer are parameterized using classes of type uvm_sequence_item. Sequences that have to be developed need to use objects of these type too. All the internals of uvm_sequence_items aren't required to be known for developing tests. All we need to know is that the stimulus carrying objects in a UVM-based testbench are required to be of type uvm_sequence_item. So, our transaction class type: core_cache_if has been developed by extending uvm_sequence_item.

5.3 Creating uvm_sequence:

Sequences are the stimulus generating engines of tests. They can be used and reused in many different ways to generate different kinds of stimulus. Each test can have multiple sequences and can interact with the driver parallely. uvm_sequences are the components that communicate the sequence items to the driver.

uvm_sequence is inherited from uvm_sequence_item. It is a parameterized class with the parameter being the type of uvm_sequence_item to be generated. Figure 5.5 is an example of an sequence class being defined.

uvm_sequence class has an internal uvm_sequencer. This object is named m_sequencer. The stimulus generation is handled by two tasks in the uvm_sequence: body() task and start() task.

The start() task is the interface to the outside world that takes in a uvm_sequencer

```plaintext
class reset_sequence extends uvm_sequence #(core_cache_if);
  `uvm_object_utils(reset_sequence);
```

Figure 5.5 sequence class definition
5.3. CREATING UVM_SEQUENCE: CHAPTER 5. ADDING A TEST

as an argument. This sequencer argument should have the same type as the parameter as the uvm_sequence itself has. The sequencer received as argument is connected to the sequence’s internal m_sequencer, thus establishing a connection between the internal sequencer to the outside world. The start() task then executes the body() task.

The body() task is where the stimulus generation code is to be developed. Figure 5.6 is an example code snippet of the body() task. The sequence has two tasks that use the m_sequencer: start_item() and finish_item() which are used in the body().

- start_item() is a blocking call. A completed start_item() call is an indication that the driver is ready to receive a new sequence item. The argument to the start_item method is a sequence item.

- The finish_item is also a blocking call. The argument to finish_item() is the generated sequence item that should be sent to the driver. A completed finish_item call is an indication that the sequence item was received by the driver and processed.

Steps to develop a sequence body():

1. Define a sequence_item.

2. Instantiate the item.

3. start_item() is a blocking method which waits for the driver to signal that it is ready to receive a new item from this sequence.

```task body();
    core_cache_if reset_command;

    reset_command = new("reset_command");

    start_item(reset_command);
    if(!reset_command.randomize())
        `uvm_fatal("RESET_SEQUENCE", "Failed to randomize input transaction");
    reset_command.reset = 1;
    finish_item(reset_command);
endtask // body
```

**Figure 5.6** body() task example
5.3. CREATING UVM_SEQUENCE:

4. Set the values of the sequence item. In our reset_sequence we've done this by randomizing the item, and then by setting the reset bit high for the driver to reset the DUT.

5. finish_item() is also a blocking routine which waits for the driver to complete its side of the operation.

6. Repeat steps 5 to 7 as many times as needed. In our example, reset_sequence doesn't need to repeat them.

The sequences once created can be used in multiple tests. They can also be used in other sequences. If we intend to use the sequencer inside another sequence class, we call the start() task by passing the m_sequencer. Figure 5.7 is the body() of the parallel_line_sequence. A line_load_sequence and line_store_sequence is instantiated in the constructor and are spawned as two different threads in the body() task.

Defining sequences gives us many opportunities to reuse the stimulus generation mechanisms for many tests. Steps for creating sequences:

1. Create a new class by extending uvm_sequence. Register it with Factory. Override the constructor.

2. Override the body() task.

```verilog
define new(string name = "");
    super.new(name);
    load_seq = line_load_sequence::type_id::create("load_seq");
    store_seq = line_store_sequence::type_id::create("store_seq");
endfunction // new

task body();
    fork
        load_seq.start(m_sequencer);
        store_seq.start(m_sequencer);
    join
endtask // body
```

Figure 5.7 Sequences using other sequences
5.4 Setting the value of the sequence_items:

The crux of the sequence stimulus is the values of the sequence_items that we set between the start_item() and finish_item() calls. The sequence_items we use are of the type core_cache_if. Table 5.1 gives the description of the core_cache_if class, so that the reader can set the values according to the functionality they want to generate in the DUT.

Table 5.1 core_cache_if class details

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>1</td>
<td>reset</td>
<td>Reset the DUT</td>
</tr>
<tr>
<td>static</td>
<td></td>
<td>‘DCACHE_TAG_BITS</td>
<td>prev_load_tag</td>
</tr>
<tr>
<td>static</td>
<td></td>
<td>‘DCACHE_INDEX-_BITS</td>
<td>prev_load_index</td>
</tr>
<tr>
<td>static</td>
<td></td>
<td>‘DCACHE_BYTES-_IN_LINE_LOG</td>
<td>prev_load_offset</td>
</tr>
<tr>
<td>static</td>
<td></td>
<td>‘DCACHE_TAG_BITS</td>
<td>prev_store_tag</td>
</tr>
<tr>
<td>static</td>
<td></td>
<td>‘DCACHE_INDEX-_BITS</td>
<td>prev_store_index</td>
</tr>
<tr>
<td>static</td>
<td></td>
<td>‘DCACHE_BYTES-_IN_LINE_LOG</td>
<td>prev_store_offset</td>
</tr>
<tr>
<td>rand</td>
<td>1</td>
<td>load_en</td>
<td>Send a load command</td>
</tr>
<tr>
<td>rand</td>
<td></td>
<td>‘DCACHE_BYTES-_IN_LINE_LOG</td>
<td>load_address_offset</td>
</tr>
<tr>
<td>rand</td>
<td></td>
<td>‘DCACHE_INDEX-_BITS</td>
<td>load_address_index</td>
</tr>
<tr>
<td>rand</td>
<td></td>
<td>‘DCACHE_TAG_BITS</td>
<td>load_address_tag</td>
</tr>
</tbody>
</table>
5.4. **SETTING THE VALUE OF THE SEQUENCE_ITEMS:**

<table>
<thead>
<tr>
<th>rand bit</th>
<th>‘LDST_TYPES_LOG'</th>
<th>load_size</th>
<th>Size of the load command to be sent</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand bit</td>
<td>1</td>
<td>store_en</td>
<td>Send a store command</td>
</tr>
<tr>
<td>rand bit</td>
<td>‘DCACHE_BYTES_-IN_LINE_LOG'</td>
<td>store_address_-offset</td>
<td>Offset of the store address</td>
</tr>
<tr>
<td>rand bit</td>
<td>‘DCACHE_INDEX_-BITS'</td>
<td>store_address_-index</td>
<td>Index of the store address</td>
</tr>
<tr>
<td>rand bit</td>
<td>‘DCACHE_TAG_BITS'</td>
<td>store_address_-tag</td>
<td>Tag of the store address</td>
</tr>
<tr>
<td>rand bit</td>
<td>‘LDST_TYPES_LOG'</td>
<td>store_size</td>
<td>Size of the store command</td>
</tr>
<tr>
<td>rand bit</td>
<td>‘SIZE_DATA'</td>
<td>store_data</td>
<td>The data to be sent to be stored</td>
</tr>
<tr>
<td>rand bit</td>
<td>1</td>
<td>retry</td>
<td>Should we retry the command. If this is a load command and it misses, this will retry till we get a hit. If this is a store and if the store is stalled, we wait till we can send this command</td>
</tr>
<tr>
<td>rand int</td>
<td>Default int size</td>
<td>delay_between_--commands</td>
<td>Delay to be kept after we send this command is successfully sent</td>
</tr>
<tr>
<td>rand int</td>
<td>Default int size</td>
<td>delay_for_load_--reply</td>
<td>If we are sending a load, delay between this and when we get a cache fill from lower hierarchy</td>
</tr>
<tr>
<td>rand bit</td>
<td>‘DCACHE_LINE_SIZE'</td>
<td>mem_load_data</td>
<td>For a load cache fill, what data should be lower hierarchy send</td>
</tr>
<tr>
<td>rand int</td>
<td>Default int size</td>
<td>delay_for_--store_reply</td>
<td>If this is a store command, what should be the delay before we get a store complete</td>
</tr>
</tbody>
</table>
### 5.5 Creating new tests

Now we know how to create sequences, we can see how to use them to create tests. All the tests extend the base_test class that we created earlier. Each test inherits the base_test’s sequencer_h object and its connection made with the driver along with it. Therefore, every test need only concentrate on the objective of generating stimulus rather than making connections.

As the base_test already covers the connections part, the only thing we need to add to new tests is the run_phase. This is where we will be sending the stimulus in through the sequencer_h. We don’t generate the stimulus sequence items in the test directly but use uvm_sequences to do that for us. We instantiate the sequences that we want in the tests run_phase and call their start() task by sending in the sequencer objects. As an example,

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand bit</td>
<td>1</td>
</tr>
<tr>
<td>invalidate_valid</td>
<td>If this is an invalidate command</td>
</tr>
<tr>
<td>rand bit</td>
<td>‘DCACHE_INDEX_BITS</td>
</tr>
<tr>
<td>invalidate_address_index</td>
<td>Index of the address to be invalidated</td>
</tr>
<tr>
<td>rand bit</td>
<td>‘DCACHE_TAG_BITS</td>
</tr>
<tr>
<td>invalidate_address_tag</td>
<td>Tag of the address to be invalidated</td>
</tr>
<tr>
<td>rand int</td>
<td>Default int size</td>
</tr>
<tr>
<td>invalidate_delay</td>
<td>Delay after which we should send the invalidate command</td>
</tr>
<tr>
<td>rand bit</td>
<td>1</td>
</tr>
<tr>
<td>scratch_mode_enable</td>
<td>High if scratch mode is enabled</td>
</tr>
<tr>
<td>rand bit</td>
<td>‘DCACHE_INDEX_BITS + ‘DCACHE_BYTES_IN_LINE_LOG</td>
</tr>
<tr>
<td>scratch_wr_addr</td>
<td>Address if the write command is sent for scratch mode</td>
</tr>
<tr>
<td>rand bit</td>
<td>1</td>
</tr>
<tr>
<td>scratch_wr_en</td>
<td>High if data is to be written in scratch mode</td>
</tr>
<tr>
<td>rand bit</td>
<td>8</td>
</tr>
<tr>
<td>scratch_wr_data</td>
<td>Data that is to be written</td>
</tr>
</tbody>
</table>
5.5. CREATING NEW TESTS

CHAPTER 5. ADDING A TEST

we’ll take a look at the line_load_test in the code distribution. The run_phase of the line_load_test are shown in figure 5.8. We have instantiated two objects here, both of which are uvm_sequences. We have called the start routine for the objects by passing the sequencer_h object to them. We have used the raise_object and drop_object routines which are an indication to the UVM environment that the test would like the run_phase to continue. More details on phases and objections can be found at section 4.1.4. That is all that is needed in the new tests.

As we can have multiple sequences in a test, there are many ways in which we can make the sequences interact with one another. In the example we saw in line_load_test, we called the sequences one after the other. Which means that the reset sequence will finish execution first, then the line load sequence will start executing.

We can also run multiple sequences in parallel. An example of sequences running in parallel can be found in stride_test’s run_phase shown in figure 5.9. This test uses three sequences: reset_sequence, stride_load_sequence, stride_store_sequence. It instantiates them in the run_phase. When calling the start() routines of the sequences, the test first completes the reset_sequence and then spawns different threads for the start() routines of stride_load_sequence and stride_store_sequence, effectively running the stimulus in parallel. The start_item() and finish_item() routines in each of the sequences interact with the driver seq_item_port’s get_next_item() and item_done() routines to decide which

```
task run_phase(uvm_phase phase);
    reset_sequence reset_seq;
    line_load_sequence load_seq;

    reset_seq = reset_sequence::type_id::create("reset_seq");
    load_seq = line_load_sequence::type_id::create("load_seq");

    phase.raise_object(this);
    reset_seq.start(sequencer_h);
    load_seq.start(sequencer_h);

    phase.drop_object(this);
endtask // run_phase
```

Figure 5.8 run_phase() of a test
5.6. One-time compilation of the new sequence and the test:

Once we have developed a new test and sequences associated with it, we need to compile them for them to be part of the test-suite. We need to add the test and sequence files in the dcache_tb_package.sv file. The package is shown in the code snippet 5.11. The compilation steps can be found in the README file of the code distribution. We only need to compile them once, after which we can switch between tests. The position where the files are added to the compilation package won't matter provided that the correct compilation order is maintained. Recommended practice is to add the sequence files to the end of the "// Test sequences" section and the test files at the end of the "// Tests" section.
Figure 5.10 UVM_TEST
5.6. ONE-TIME COMPILATION OF THE NEW SEQUENCE

Figure 5.11 dcache_tb_package

```vhdl
// Transaction classes between testbench components
include "tb/cache_core.sv"
include "tb/cache_in.sv"
include "tb/cache_core_out.sv"
include "tb/cache_memory_out.sv"
include "tb/coverage_trans.sv"

// Testbench components
include "tb/scoreboard.sv"
include "tb/monitor.sv"
include "tb/driver.sv"
include "tb/checker.sv"
include "tb/dcache_coverage.sv"
include "tb/env.sv"

// Test sequences
include "tb/reset_sequence.sv"
include "tb/random_sequence.sv"
include "tb/line_load_sequence.sv"
include "tb/line_store_sequence.sv"
include "tb/parallel_line_sequence.sv"
include "tb/mshr_data_line_sequence.sv"
include "tb/stress_test_sequence.sv"
include "tb/stride_load_sequence.sv"
include "tb/stride_store_sequence.sv"
include "tb/store_buffer_size_sequence.sv"
include "tb/mshr_data_sequence.sv"

// Tests
include "tb/base_test.sv"
include "tb/random_test.sv"
include "tb/line_load_test.sv"
include "tb/line_store_test.sv"
include "tb/parallel_line_test.sv"
include "tb/stress_test.sv"
include "tb/stride_test.sv"
include "tb/store_buffer_size_test.sv"
include "tb/mshr_data_test.sv"
```
5.7 Running a test, switching between tests:

Now that we've added tests and sequences to the package, we can run the test. We need to compile the package using run.do script in the code distribution. The knob for changing the test in the simulation is the UVM_TESTNAME argument. The arguments for running the tests are given with the simulation command. The arguments are:

+UVM_TESTNAME=<name_of_the_test_class>

It is this simple. Changing the +UVM_TESTNAME argument for each simulation, we can switch between running different tests seamlessly.

5.8 Steps for adding a test:

Concluding the discussion in this chapter, the steps to add a new test are as follows:

1. Create new sequences for the test if needed.
2. Create a new test class by extending the base_test.
3. Edit the run_phase of the class to either define, initiate and start the sequences.
4. Add the test and sequence files to the dcache_tb_package.sv file
5. Compile the test suite.
This chapter describes the coverage and the tests that were developed to test the cache and achieve the coverage defined. The coverage metric is what defines the completeness of the verification process. The stimulus definition decides if we will reach our target coverage. The stimulus is to be changed with time to reach our coverage goal. It is important to define our coverage comprehensively so that we get confidence that our verification plan is testing all possible functionality of the DUT. In this chapter, we first talk about the coverage metric that we are using for verification. We also discuss the stimulus used for testing.
6.1 Coverage:

We concentrate on each of the features and the expected exercising of these features to develop a coverage metric. Usually we look at the possible inputs, possible outputs and the valid ways in which we can get cross coverage between the inputs and outputs. We then focus on the internal workings of each of the features. Coverage needs to concentrate on gathering information not data[1]. We need not gather all the data input values that came in, but do we need to gather information about whether every event that could have occurred in the internal functioning of the cache has occurred. In this chapter we define the coverage metric in terms of statements rather than SystemVerilog code. The dcache_coverage class in the code distribution has the SystemVerilog covergroups defined, instantiated and sampled in it.

6.1.1 Load coverage:

Loads received from the processor core can target different cache lines and different offsets in the same line. The interface also allows for loads of different sizes. The load can result in either a load hit or a load miss. We would first like to cover all possible:

1. Load requests are received for all possible cache lines
2. Load requests are received for all word offsets in a cache line
3. Load requests are received for all possible sizes in the same word
4. Load requests are received for all byte offsets in each word
5. Load requests result in hit.
6. Load requests result in a miss.
7. Load requests received for a valid cache line but with a different address tag is received. It would seem that this is similar to 6. since both result in a miss, but this is a specific case. While 6. concentrates on all possible misses, 7. wants to cover a specific event.
8. Cross coverage of all of these together. Which would mean that load requests are received by all cache lines, all possible word offsets in each cache line, all possible byte offsets in each word, all possible load sizes in a word, and result in a hit or a miss.

An important thing that the above coverage metric did not include is the internals of the load results. We have store/load forwarding in the cache which means that the load can be a hit or miss for multiple scenarios occurring in the cache line and the store buffer. A load can be a hit because it was present in the cache line or because it was in the store buffer or present in both. A load can be a miss because address was not present in either cache line or store buffer, or because address was present in the cache line but a partial data was present in the store buffer. We need to ensure that these conditions are generated by our stimulus for testing.

1. For a load request, the address is present in the cache line and not in the store buffer
2. For a load request, the address is present in the cache line and in the store buffer
3. For a load request, the address is present in the cache line and a partial version of requested address is present in the store buffer
4. For a load request, the address is not present in the cache line and not present in the store buffer
5. For a load request, the address is not present in the cache line and present in the store buffer
6. For a load request, the address is not present in the cache line and a partial version of requested address is present in the store buffer
7. Cross the above conditions with all cache lines to ensure that each cache line is tested for the generation of the above conditions.

The cache has an Miss Status Handling Register (MSHR) which ensures that the cache interface is non-blocking and that multiple load requests aren't forwarded to the lower hierarchy of the memory. A load miss may result in either the load being forwarded to the lower hierarchy or not being forwarded. This depends on the contents of the MSHR. If a
6.1. COVERAGE: CHAPTER 6. COVERAGE AND STIMULUS GENERATION

requested load address is present in the cache when the MSHR is valid, the result should be a hit. In these example, we are strictly considering the different possible values in the MSHR when we receive a load request. And we want to ensure that these scenarios occur when we are testing the cache.

1. A load request, different address than the one in MSHR, is received and results in a hit.

2. A load request, different address than the one in MSHR, is received and results in a miss.

3. Load request to same cache line as the one in MSHR and the same tag is received. Basically the same load that filled the MSHR.

4. Load request to same cache line as in the MSHR but different tag is received.

6.1.2 Store coverage:

Like we discussed all types of inputs that can be received for load, we have similar coverage groups for the store requests. The basic coverage metric for stores will be as follow:

1. Store requests to all cache lines

2. Store requests to all possible word offsets in a cache line.

3. Store requests for all possible sizes in a word offset.

4. Store requests to all possible byte offsets in a word.

5. Cross coverage between all the above.

An important part of stores is that if a store completes to a address which is already in the cache line, the data is merged with the existing cache line. The test for this particular scenario would be if the store results in a hit. If the address isn’t present in the cache line but is part of the MSHR, the MSHR data is updated so that it gets merged when the cache line is filled. The way we can check this is by checking the MSHR byte enable register values. Here's how we check if these scenarios have occurred:
6.1. COVERAGE: CHAPTER 6. COVERAGE AND STIMULUS GENERATION

1. Store results in a hit

2. Store results in a miss

3. Cross for hit/miss for all possible cases. So hits and miss for all lines, all sizes, all word and byte offsets.

4. MSHR byte-enable is > 0. Which means that it is valid and has data stored in the MSHR by a completed store.

The stores also go through the store buffer. Received store requests are forwarded immediately and the store buffer keeps track of which stores have completed. Coverage information should be collected with respect to the store buffer. We could check if the store buffer was filled with multiple depths, instead we can use the full and empty flags. If the store buffer goes from empty to full and back, we can be sure that it went through all possible depths.

1. Store buffer is empty

2. Store buffer is full

3. Store buffer goes from empty -> full -> empty

6.1.3 MSHR coverage:

We have covered a lot of MSHR functionality in the load and store covergroups. We’ve covered different load requests when the MSHR is valid. We’ve covered the buffering of store data in the MSHR data buffer. In covergroups related to MSHR we consider all possibilities for the MSHR values. As we saw in the store covergroups, the store requests affect the MSHR data and the byte enable fields. We’re concentrating on all possible combinations of bytes the MSHR can have store in its MSHR data structure. We should gather data for conditions where a store request should result in the MSHR caching data and cases where it shouldn’t. Following are the coverpoints related to MSHR:

1. MSHR index corresponds to all possible cache lines

2. MSHR byte enable takes on all possible values
3. Cross coverage between the above

4. Store requests to the same cache line, but different tag as in MSHR.

5. Store requests to the same cache line, same tag in the MSHR.

6. Store requests to the different cache line, same tag in the MSHR.

7. Store requests to the different cache line, different tag in the MSHR.

8. MSHR data byte enable was valid and the store, when completed, results in a hit. This means that the cache line was filled before the store completed.

9. MSHR data byte enable was valid and the store, when completed, results in a miss. This means that the store completed before the cache line was filled.

6.1.4 State-space coverage:

We also need to cover the state-space for each cache line[1]. Cache lines have state bits associated with them which can take various values depending on the type of the cache. For a complex cache-coherence protocol we can have various states that a cache line can be in. Since our target cache is a write-through cache with an invalidation only protocol, there are only two states that the cache lines can have is valid and invalid. The coverage metric for this line would not be as complicated, but needs to be covered nonetheless. Also, the covergroups that we've discussed thus far deal with the interface between the cache and the processor core. The state space coverage in a way is related to the other interface: the lower-hierarchy interface. We can get 3 types of signals from the lower hierarchy interface.

- Cache line fill in response to a load
- store complete signal in response to a store
- invalidation from the lower hierarchy.

There are a few conditions we are not testing with respect to these signals:
6.1. COVERAGE: CHAPTER 6. COVERAGE AND STIMULUS GENERATION

- A load request to the lower-hierarchy/memory always generates a cache fill response. Not receiving the cache fill isn't a valid condition. We won't test for the condition of not receiving cache-fill response and won't cover it either.

- A store complete signal is received from the lower-hierarchy/memory every time a store request is sent. Not receiving it isn't a valid case so we won't have to test and cover it.

For the rest of the interface activity the following cover-points are considered:

1. A cache fill is received by all cache lines
2. A invalidation request is received by all cache lines

State-space coverage of the cache is related to these signals received from the lower hierarchy in the following way:

- Receiving a cache line fill from the lower hierarchy means that the cache state will go to valid, irrespective of its previous state. It may have been already valid, in which case the address cached in the line has changed, we get a valid -> valid transition. Otherwise, it will go from invalid -> valid.

- Receiving an invalidation for a cache-line means that the cache state will go to invalid, irrespective of the previous state. If it was invalid, we get a invalid -> invalid transition. Otherwise we get a valid->invalid transition.

6.1.5 Delays:

Delays are an important part of verification. Different delays can cause the internals of the design to interact in different ways. Using different delays within the valid limits can lead to bugs becoming more apparent. In fact, as we will discuss later, randomized delays did reveal a few bugs in testing. Following are the delays that we measure coverage for:

- Delay between the load request and the corresponding cache fill response from lower hierarchy.
• Delay between store request and store complete from the lower hierarchy corresponding to that store request.

• Delay between the cache fill from lower hierarchy and a new load request (same or different from the existing one in the MSHR).

• Delay between the store complete from lower hierarchy and a store request.

We could add the measurement of these delays in either the coverage class or in the BFM. As the objective of our testbench was to be a transaction-level testbench, we measured the delays in the BFM and sent them to the coverage class to sample them. Alternatively, the delay metric can still be measured in the coverage class itself. But then the class will have to change when the signals change significantly.

6.1.6 Scratch mode coverage:

In the scratch mode, the cache acts like a scratch-pad and there are no miss results for either load and store. The tags for either the store or load addresses aren't used for storing or retrieving data hence they aren't considered in the coverage metric. Signals that are specifically related to the scratch-mode are considered in the coverage. For store commands:

1. Stores received by all cache lines when scratch-mode is enabled

2. Stores received with all possible word-offsets when scratch-mode is enabled.

3. Stores received with all sizes when scratch-mode is enabled.

4. Cross coverage between the above, i.e., stores received for all cache lines, with all possible word-offsets in each cache line and with all possible sizes for each cache line.

For load commands:

1. Loads received by all cache-lines when scratch-mode is enabled.

2. Loads received for all possible word offsets when scratch-mode is enabled.
3. Loads received with all possible sizes when scratch-mode is enabled.

4. Cross coverage between the above, i.e., loads are received for all cache lines, for all possible word offsets in each line with all possible sizes for each word.

For scratch mode signals:

1. Scratch write command received for all cache lines
2. Scratch write commands received for all bytes in a line
3. Cross coverage between 1. and 2., write commands received by all bytes in each line of the cache.

6.2 Stimulus generation:

Discussing the coverage metric before the stimulus helps in our understanding of the types of stimulus that should be used. As we already have concentrated on the types of conditions that we want to cover, we can come up with tests in a more efficient way. We would also not be repeating tests if we know that we’ve already developed a test to cover a feature. More tests were added in response to certain cover points not being met. An important observation from verification of the cache is that, while random stimulus is very useful, directed test cases with some randomness in the data are also very important for reaching our coverage goal. UVM sequences are used to develop these tests. A few tests were developed for OOP testbench too for comparison to the UVM tests. Each section describes the sequence used for the test. A common sequence used for all the tests is the reset_sequence, which as the name suggests resets the cache and brings the DUT to a known state.

6.2.1 Line test. Load, store and parallel:

A line test is the most basic test that should be run when verifying a cache. We are testing each line sequentially to check if it satisfies the basic functionality of the cache, i.e., exploiting spatial and temporal locality. By sending transactions to each line, we are trying to cover
the load and store request for each line coverpoint. We also hope to cover the miss/hit results coverpoint for both load and store. The line_load_sequence, line_store_sequence and parallel_line_sequence classes in the code distribution are the three sequences used for these tests. The description of the sequences and the tests is in the following sub-sections.

6.2.1.1 line_load_sequence:

1. Start with the first cache line (0th line), send a load request to the line. Retry till we get a hit.

2. Send a load request to same line, with the same tag, with randomized word and byte offset, randomized size. Retry till we get a hit. This should result in a hit since the previous load should have filled the line. Wait for the transaction to complete.

3. Send a load request to same line, with a different tag, with randomized word and byte offset, randomized size. Retry till we get a hit. Wait for the transaction to complete.

4. Repeat the steps 1) through 3) for all cache lines.

5. Repeat step 4) multiple times.

6.2.1.2 line_store_sequence:

1. Start with the first cache line (0th line), send a store request to the line, with a random tag. If the store buffer is full, wait till it has space. Wait for the transaction to complete.

2. Repeat step 1) 2 times for the same cache line. This is done to stay consistent with the line_load_sequence.

3. Repeat the steps 1) & 2) for all cache lines.

4. Repeat step 3) multiple times.

6.2.1.3 parallel_line_sequence:

This sequence does not generate commands of its own, instead uses the line_load_sequence and line_store_sequence to generate stimulus.
6.2. STIMULUS GENERATION: CHAPTER 6. COVERAGE AND STIMULUS GENERATION

1. Instantiate the line_load_sequence and the line_store_sequence.
2. Run them in parallel as two threads.

### 6.2.1.4 line_load_test:

The line_load_test uses the reset_sequence and the line_load_sequence for generating the stimulus.

1. Reset the DUT using reset_sequence.
2. Run the line_load_sequence.

### 6.2.1.5 line_store_test:

The test uses reset_sequence and line_store_sequence to generate stimulus.

1. Reset the DUT
2. Run the line_store_sequence.

### 6.2.1.6 parallel_line_test:

The test uses the reset_sequence and parallel_line_sequence to generate stimulus.

1. Reset the DUT
2. Run the parallel_line_sequence.

### 6.2.2 Stride test:

The stride test is the logical succession of the line test. Instead of accessing the cache lines sequentially, we access them in strides that are randomly selected. While, this test is trying to cover the coverpoints similar to what we covered with the line test, we hope the stride test could test for conditions we didn't envision in the line test. We have two sequences: stride_load_sequence and stride_store_sequence.
6.2.2.1 **stride_load_sequence:**

1. Randomly select a stride.

2. Randomize a transaction to get a random cache line and tag. Modify the transaction to a load command. Send the transaction. Wait for it to complete.

3. Randomize a transaction. Add stride to the previous index. Modify it to be a load command. Send transaction and wait for it to complete.

4. Repeat step 3) multiple times to be sure that all cache lines have been covered.

6.2.2.2 **stride_store_sequence:**

1. Randomly select a stride.

2. Randomize a transaction to get a random cache line and tag. Modify the transaction to a store command. Send the transaction. Wait for it to complete.

3. Randomize a transaction. Add stride to the previous index. Modify it to be a store command. Send transaction and wait for it to complete.

4. Repeat step 3) multiple times.

6.2.2.3 **stride_test:**

The stride_test uses reset_sequence, stride_load_sequence and stride_store_sequence to generate stimulus.

1. Instantiate the reset_sequence, stride_load_sequence and stride_store_sequence

2. Reset the DUT

3. Run the stride_load_sequence and stride_store_sequence in parallel.
6.2.3 MSHR data test:

The MSHR has a feature to store the data in case the cache receives store commands to the cache line the MSHR corresponds to. The MSHR can store a whole cache line while it waits for the line to be filled from the lower memory hierarchy. When the cache line is received the MSHR merges the received data with the data it has stored. We are trying to cover these features using this test. Roughly we are targeting the MSHR covergroup along with a few coverpoints in the load coverage related to MSHR. As a measure to stress this feature, the MSHR data test sends multiple store commands to a specific pending cache line to fill its MSHR data buffer with as much data possible. The mshr_data_line_sequence and mshr_data_sequence are developed for this test.

6.2.3.1 mshr_data_line_sequence:

This sequence isn't designed to be a standalone sequence, and should be used in collaboration with an already running sequence.

1. Send a load command to the DUT using the last used load index and a specific tag. This is done to generate a known state. Retry till the load command doesn't return a hit. Wait till the transaction completes.

2. Send another load command for the same index and a different tag. This is bound to result in a miss since the cache line had loaded the previous data. For this transaction, give a large delay for the lower hierarchy to respond. We've used 500 cycles. This results in a large time period in which the MSHR has a valid value, but the cache line isn't filled. This is a ideal time for store commands to fill up the MSHR data buffer. Don't retry the load command.

3. Send multiple store commands to the same cache line, some with the cache line that missed, some for the cache line already present. Wait for a transaction to complete before sending another.

4. Now send load commands to each of the words in the cache line to read out the value stored in it. We expect it to be the data received from the lower memory hierarchy merged with the one present in MSHR data buffer.
6.2.3.2  mshr_data_sequence:

The mshr_data_line_sequence focuses on a single line. We need to extend that to work for
all lines.

1. Send a load command to the 0th cache line. Wait for the transaction to complete.

2. Instantiate the mshr_data_line_sequence.

3. Start the mshr_data_line_sequence for the line.

4. Repeat 1) through 3) for every line in the cache.

6.2.3.3  mshr_data_test:

1. Instantiate the reset_sequence and mshr_data_sequence.

2. Reset the DUT

3. Start the mshr_data_sequence.

6.2.4  Stress test:

The stress test is designed to stress a particular line in the cache with bursts of random
commands from the processor core and invalidation requests. We also test the MSHR data
like we did in the mshr_data_test after we done sending random commands. We define the
stress_test_sequence for the test. We can achieve state-space coverage using this test pretty
soon since it targets a single line for multiple commands including invalidation commands.
Also, we can achieve some delay coverage since many expected delay conditions will be
present. The MSHR data sequence is run again to reveal any bugs after stressing a single
line.

6.2.4.1  stress_test_sequence:

1. Send a random, but non-invalidate, transaction to 0th cache line. Wait for the trans-
action to complete.
2. Disable the non-invalidate constraint, send random transaction to DUT for the same line. It can be load, store, invalidate, random delay, random tag, random size. Wait for the transaction to complete.

3. Repeat step 2) multiple times. Currently repeated 200 times. This is simple to change in the sequence.

4. Instantiate the mshr_data_line_sequence. Start the sequence.

5. Repeat steps 1) through 4) for every line in the cache.

### 6.2.4.2 stress_test:

1. Instantiate reset_sequence and stress_test_sequence.

2. Reset the DUT

3. Start the stress_test_sequence.

### 6.2.5 Random stimulus:

In this test we send randomized transactions to the DUT. This is a simple test and sequence to write but is very powerful in the sense that it can reveal issues within the design that we as verification engineers didn't look for. We define a random_sequence for this test. We have also designed the test for the OOP-testbench for comparison. This test mostly serves the purpose of revealing bugs that we haven't envisioned a condition for.

#### 6.2.5.1 random_sequence:

1. Instantiate a transaction. Randomize it.

2. Send it to DUT. Wait for it to complete.

3. Repeat steps 1) and 2) multiple times. The number of commands to be sent is simple to change in the sequence.
6.2.5.2 random_test:

1. Instantiate the reset_sequence and the random_sequence
2. Reset the DUT
3. Start the random_sequence.

6.2.6 Scratch mode test:

We test the scratch mode features of the cache in this test. Both load and store are valid for
scratch mode. In addition to that the scratch mode has a write interface which can write a
byte at a time. Since the scratch mode acts as a scratch-pad, loads and stores always result
in a hit. We need to ensure that the load requests are always sent to cache-lines that have
valid data already stored in them. For this purpose, in our sequence we first send store
commands to all the cache lines and words before starting out with the test.

6.2.6.1 scratch_mode_sequence:

1. Send a transaction with only the scratch mode enable signal high. This ensures that
   all future transactions, even if they are in the very next cycle will be considered in
   scratch mode.
2. Send store commands to each word in each line with size: word to fill all the cache
   lines with valid data
3. Send multiple random commands by disabling the scratch mode disable constraint.
   Currently we are sending a million commands. But this can change in future if the
   coverage isn't getting met.

6.2.6.2 scratch_mode_test:

1. Instantiate the reset_sequence and scratch_mode_sequence.
2. Reset the DUT
3. Start the scratch_mode_sequence.
Starting out with a basic Object Oriented testbench, we've discussed the development of the UVM-based verification suite. We discussed the various features of the UVM base-class library. In this chapter we will be discussing two results from the UVM-development effort.

First, we will be comparing the effort needed to add/change various features of the testbench between the UVM-based testbench and OOP-based testbench. We will try adding a new test to both the testbenches. We will also compare the effort needed to change a feature in the scoreboard. These will give us an idea of the advantage that UVM can provide in terms of re-usability.

Secondly, we will also discuss the bugs that were found during the verification phase. Important thing to note here is the identification of an observation as a bug depends on the specification. If a specification isn't followed, it is defined as a bug. The specifications, even when clearly written, may not be perfect and changes to the specifications are fairly common. A bug may not even result in faulty operation, the cache may perform fine without actually affecting the execution of any program on the core.
7.1 Comparing the two testbenches

We'll discuss two changes that may occur fairly commonly in any verification plan and will discuss the results, taking into account the effort for each change, flexibility and reusability of the organization.

7.1.1 Adding a new test:

The existing verification suite has the stride test, which sends load/store requests to sequential cache lines, keeping a constant stride between the cache line accessed. The test intends to test the functionality of the cache when different lines are accessed and/or paralelly filled with new data. Table ?? documents the changes made to the OOP-testbench and to the UVM suite to add this test.

The important thing to compare here is not only the amount of effort needed to add a test to the framework, but also the impact and further re-usability this added to the suite.

In the UVM suite: In adding the stride test, we defined two different sequences: a stride_load_sequence and a stride_store_sequence each handling only one specific operation. The stride test used these two sequences and runs them in parallel to create a test that sends either a load or a store command. These two sequences can be used for other tests too without any impact on the existing stride_test.

As an example, we've at the very beginning defined a reset sequence, which resets the DUT. We've been re-using this sequence in every test, so we don't have to re-write code for doing the very simple operation of resetting the cache and setting all the inputs to a

---

**Table 7.1 Comparison for adding tests**

<table>
<thead>
<tr>
<th>Files changed</th>
<th>UVM-based testbench</th>
<th>OOP-based testbench</th>
</tr>
</thead>
<tbody>
<tr>
<td>added stride_load_sequence.sv</td>
<td>added stride_store_sequence.sv</td>
<td>added stride_tester.sv (extends class tester)</td>
</tr>
<tr>
<td>added stride_store_sequence.sv</td>
<td>added stride_test.sv</td>
<td>modified DCache_testbench.sv</td>
</tr>
</tbody>
</table>

Comparing added 141 lines

added 37 lines,
modified a key testbench component
non-asserted state.

Also, adding this test has in no way impacted the existing tests in the suite. We can switch to any previous test and run it by just providing a different directive in "+UVM_TESTNAME" argument while running vsim next time.

When adding the stride test to the OOP-testbench, we modified two key components of the testbench: the tester class and the testbench class. Going back to any other test would involve changing either both these classes again or changing testbench at the very least. So running multiple tests will need change to the top-hierarchy of the testbench and compilation of the testbench for every change. Moreover since we had to extend the tester class to achieve the stride test, we kept most of the functionality of the tester class, for example we could not change the number of commands that we can send. For achieving this, we would have to change the top module, which is a huge change for a simple task of changing the number of commands sent in. The stride test that we defined also chose out of load and store randomly and started at the same index for both. UVM sequences gave us much better flexibility in fine tuning our test.

Comparison between OOP and UVM for adding a new test:

- **Effort**: Comparable. While the number of lines may give an impression that UVM sequences were more difficult to develop than OOP tests, but the complexity for both of them is comparable. Moreover we'll see the advantages in the next two points.

- **Reusability**: UVM gave us much better reusability for the tests. We were able to add the new test without making any change to the testbench structure. We were able to retain the old tests and could run them without making any changes. We also used the already defined reset sequence, so we did not repeat code. The sequences defined for this test can also be used for future tests. In OOP testbench, we did extend the already defined tester class. But we had to modify the testbench class which is a key component of the hierarchy. UVM did the same job with no modifications to the existing testbench structure.

- **Flexibility**: UVM sequences gave us much better control over the smaller aspects of the test, like the index from which we would like to use the load and store commands. The strides for each load and store could be chosen differently. The number of tests
could be changed for each sequence without affected other sequences. To switch to an older test in the OOP-testbench, we'll have to modify testbench class again. To change the number of commands to be sent, we'll have to modify the top module.

Clearly UVM gives us much more flexibility with no modifications to the existing testbench.

### 7.1.2 Changing a feature in the scoreboard:

A feature that isn't present in the cache now is: invalidation of data present in the store buffer. Currently, the coherence protocol is an invalidate-only protocol. Reception of an invalidation request to a data that is present in the store buffer is not a valid case for this cache. But if in the future we need to add this feature, the effort needed to add it to the scoreboard for the two types of testbenches will be the same. This is because the scoreboard was designed according to the specifications. Since, the specifications have changed the scoreboard implementation will have to change.

### 7.1.3 Comments

From the two sub-sections above and the UVM features discussed in the previous chapters, we can conclude that UVM makes connections between different aspects of the testbench simple and reusable. We get more flexibility for defining tests. The existing testbench structure is robust and does not need to change when adding tests.

Whereas, when it comes to changing features of the DUT itself the UVM-based testbench will have to undergo almost the same amount of changes as the OOP-based testbench. This is expected since a testbench cannot remain immune to changes in the features of the DUT.

### 7.2 Bugs revealed by the verification suite

#### 7.2.1 Inconsistency in load-forwarding:

This bug was revealed by the line_load_test in the verification suite. If a load request from the core/upper-hierarchy misses, and the MSHR is empty, the cache is expected to forward
7.2. BUGS REVEALED BY THE VERIFICATION SUITE

Figure 7.1 Load request to same line, different tag. Request Forwarded

load requests to the lower-hierarchy. In some cases, the MSHR may not be empty for the current cycle, but may have valid fill data from the lower-hierarchy because of which the MSHR will get empty in the next cycle. In such cases, the cache forwards the load request if for a different address from the one being filled. There is some inconsistency in the way the cache behaves if the new load request address is in the same cache line as the one being re-filled.

- Figure 7.1 shows a case where the load-request is forwarded.
- Figure 7.2 shows a case where the load-request isn't forwarded.

7.2.2 Load response behavior on partial hit in store buffer:

This bug was revealed in the stress_test. Cache implements store-load forwarding to send data stored in the store buffer to a load request. In cases where the load request address is partially present in the store buffer, neither we cannot send the data stored in store buffer since it isn't complete, nor can we send the data in the cache-line because it is stale. Cache is expected to send a miss in such a case. It is observed that the cache doesn't respond to a load request with either a hit or a miss if there is a partial store buffer hit.
7.2.3 **Load request forwarded twice for the same load address**

This bug was revealed by the random_test in the verification suite. The cache forwards load requests to the lower-hierarchy if a load request from the core/upper-hierarchy misses, and the MSHR is empty. The core keeps on retrying the load till it gets a hit. If the cache fill response from the lower-hierarchy is received and in the very next cycle another load request from the core for the same address is received, the load will miss since the data will be available one cycle later. In this miss, the cache isn’t expected to forward the load to the lower-hierarchy, but it does. *This bug was in a previous version of the cache and isn't present in the current version.*

7.2.4 **Store forwarding size**

This bug was revealed by the stress_test. If the store buffer has a word of data stored in it. The cache receives a load request for a data of byte size which is part of the word data stored in the store buffer, but isn’t aligned to the least significant byte of the word.

The cache should forward the data in the store buffer to the core, but instead it marks this as partial hit for the store buffer.

This issue was later resolved as the being added to the specification. While the forwarding scenario makes sense from a performance stand-point, it makes implementation of
store-load forwarding in RTL complicated. So, currently this isn't being classified as a bug, this has been added to the specification in section 2.3.5.

7.3 Coverage Achieved

Achieving 100% coverage is possible if the DUT is bug-free. Once we find a bug, the coverage achieved for the specific goal isn't considered valid. Since, we still have some load/store bugs, we cannot claim that we achieved 100% coverage. But we would like to know if the stimulus generation schemes that we used in our verification suite will be able to achieve the coverage goal. We performed a few experiments to check this.

- The `line_load_sequence` was created to achieve the coverage goal of the load features. Since the sequence revealed a bug, we cannot report the coverage achieved by the sequence as valid. But we changed our stimulus to avoid this bug and confirm that the coverage achieved has increased.
  - Before changing the stimulus, the load covergroup coverage achieved was 81%. Out of which the index coverage was 100%, offsets coverage was 100%, results coverage was 100%. Only cross coverage wasn't achieved and it was 8.8%
  - Once we changed the stimulus so that we don't hit this bug, and ran the same sequence for 1 million transactions, the cross coverage reached 58%. We specifically changed the stimulus to not hit scenarios where we encountered the bugs. So, the coverage we achieved will be more when the bug is fixed and we change the stimulus to what it originally was.

- For features which did not have bugs, e.g., scratch-mode, we were able to achieve 100% coverage. But on fixing all the bugs, the tests needs to re-run to confirm the coverage.
CHAPTER

8

CONCLUSION

We started out with the goal of exploring UVM-features by using them to verify the H3 cache\[4\]. While the functional verification for the cache was already done, we focused on achieving comprehensive coverage in our testbench. The first step to do that was to document the specifications of the cache, which we did in Chapter 2.

Our goal in using UVM was to explore the features and document the process of using the features. We also wanted to document the advantages that we get from using UVM. We developed a OOP-testbench to have a baseline to compare to. We explored the testbench in Chapter 3.

In chapter 4 we first explored the UVM features that we used to develop the verification suite. We documented the verification suite as a step-wise guide to convert the OOP-testbench to use UVM features. We hope that the reader will get an idea of how to convert any existing testbench developed in SystemVerilog to UVM by reading the chapter.

For a reader wanting to use the UVM-based verification suite that we've developed, we've added chapter 5 describing how to add tests to the existing suite of tests.
In chapter 6 we've explained a comprehensive coverage metric before coming up with tests so that we are confident will reach the coverage goal in our verification plan.

Finally comparing the UVM-based verification suite to OOP-testbench we've observed that:

- While SystemVerilog provides many object-oriented programming features to make testbench reusable, UVM builds on it by developing its own base-class library and features. These features help us in solving many problems with the existing SystemVerilog framework, like the repetition of mailbox connections for sending the same data to different classes.

- The UVM Factory provides us with a way to instantiate different objects without changing code but by changing run-time arguments. This helps us a lot in simulating different tests and switching between tests. This enables us in maintaining a suite of tests which the user can switch between seamlessly.

- The UVM configuration database provides us with a way to share information between different testbench components which is immune to the hierarchy of the testbench. We don't need to pass arguments all the way from top-level class of the testbench to the lower hierarchy components anymore. Any component can get the data that it wants from the configuration database. This means we don't need to change the top-level hierarchy of the testbench every time a component needs a different set of data.

- The Observer design pattern solves the problem of multiple observers for a single generator. It also provides future scope of adding new observers without disturbing the generator class.

- UVM agents and UVM environments create a stable hierarchy which won’t change even if we have many DUTs in a design and we would like to verify each of them individually or verify them together.

- UVM seq_item_port and uvm_sequencer separate the stimulus generation from the structure of the testbench in every way. This leads to less modification of code when adding tests.
• uvm_sequences and uvm_tests are very beneficial in reusing the stimulus already
developed and can also be used to create new tests by having the stimulus interact in
a different way. This gives us more use out of the already existing stimulus.

• For a comparable effort, (adding 3 files and 114 lines of code in UVM, rather than
modifying 1 file and adding 1 file with 37 lines of code in OOP) the user can add a
test to the suite by compiling only once and then switching between existing tests.
The stimulus already generated can be re-used in UVM, which cannot be done in the
OOP testbench.

• While we couldn't achieve 100% coverage because the DUT has a few bugs. We have
seen through experimentation that the coverage can be achieved with the developed
stimulus. For features that do not have bugs, like scratch-mode, we were able to
achieve 100% coverage.
BIBLIOGRAPHY


