D-CACHE VERIFICATION OF RISC-V OUT OF ORDER PROCESSORS

A Project Report

submitted by

T NIHARIKA

in partial fulfilment of the requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY
&
MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

JUNE 2021

THESIS CERTIFICATE

This is to certify that the thesis titled **D-CACHE VERIFICATION OF RISC-V OUT OF ORDER PROCESSORS** , submitted by **T NIHARIKA**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology & Master of Technology**, is a bona fide record of the research work done by her under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. V. Kamakoti
Research Guide
Professor
Department of Computer Science and Engineering
IIT-Madras, 600 036

Prof. Harishankar Ramachandran
Research Co-Guide
Professor
Department of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 30th June 2021

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my project advisor, Prof. V. Kamakoti for providing me this opportunity to be a part of the Shakti Labs team and Prof. Harishankar Ramachandran for being my co-guide, allowing me to a do the project in the Department of Computer Science and Engineerig. I would also like to thank my mentor, Ms. Lavanya Jagadeeswaran, Project Officer at Shakti, for her extensive guidance and support throughout the course of my project and Ms. Nitya Ranganathan, Project Officer at Shakti for providing me with resources to understand the I-class cache design and clarifying my doubts regarding the same. I would also like to thank, Divya, Project Assosciate at Shakti, for helping me in setting up the work environement and resolving technical issues. Finally, I want to thank my family and friends for their constant encouragement and support throughout my education and research.

ABSTRACT

KEYWORDS: D-Cache; RISCV; UVM; PyMTL3 model; CoCoTb testbench.

Verification of digital designs is necessary to ensure the conformance of a design to some predefined set of expectations. In this case, the predefined expectations are provided by RISC-V specifications. Each block must be verified and the entire SoC needs to be verified to ensure that there are absolutely no bugs. Thus, it becomes necessary that the verification setup is easy to modify to suit all the different blocks and can be smoothly integrated for core level integration and that the frameworks of the model and testbench are easily adaptable. Also, The test bench has to be developed in such a way that it allows us to verify the functionality of the DUT in all possible scenarios. verify the working of In this project we have attempted to show the robustness of Python PyMTL3 and CoCoTb as hardware generation and verification frameworks respectively. The reference model is entirely build in PyMTL3. It provides a variety of class libraries including Components, Delays, Queues etc. which are easy to derive and override, thus providing an efficient way of implementing hardware models whether they are functional level or cycle accurate. The verification setup is developed in CoCoTb. It provides the standards of UVM and UVM classes such as BusMonitors, Drivers, Scoreboard etc. which are compact, easy to override and without any additional baggage unlike SystemVerilog, which also makes CoCoTb much easier to adapt to. Since, these are both Python based, the interfacing becomes seamless and the open-source and wide popularity of Python itself makes them a good choice.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

AAPG	Automated Assembly Program Generator
CoCoTb	COroutine based COsimulation TestBench environment
LRU	Least Recently Used
PTW	Page Table Walker
RISC	Reduced Instruction Set Computer
ROB	Reorder Buffer
SoC	System on Chip
UVM	Universal Verification Methodology

CHAPTER 1

INTRODUCTION

The Shakti I-Class is a 64 bit-processor with performance features like out-of-order execution, branch prediction, non-blocking caches, multi-threading and deep pipeline stages. It has separate Instruction and Data Caches.

The aim of this project is to verify the working of the L1 Data Cache (D-Cache) and find potential bugs in the RTL design. As part of this project, a functional level parametric reference model (predictor) that emulates the behaviour of the D-Cache is developed. A test bench is developed following the Universal Verification Methodology (UVM), accommodating out-of-order execution and non-blocking nature of the cache design.

This report briefly touches upon the Verification Methodology and the frameworks used to develop the model and the test-bench in chapter 2. Then a detailed explanation of the reference model is provided along with its features and parameters in chapter 3. The verification setup and test bench are explained in chapter 4 along with block level verification and the core level integration of the test-bench. The test suite used and coverage analysis is provided in chapter 5. Future scope of the project is given in chapter 6.

CHAPTER 2

VERIFICATION METHODOLOGY

2.1 The Universal Verification Methodology (UVM)

The Universal Verification Methodology (UVM) is a standardized methodology for verifying integrated circuit designs. It enables faster development and reuse of verification environments. UVM has a set of class libraries defined using SystemVerilog. UVM is derived mainly from the OVM (Open Verification Methodology) and is an Accellera standard with support from various vendors like Synopsys, Cadence, Xilinx Simulator, Aldec etc.

The UVM class library provides and generic utilities like configuration databases, component hierarchy and data automation features (like packing, copy, compare etc.). Every component is assigned a specific role and the component object can be derived from the corresponding class and configured to suit the needs of the particular application. Fig. 2.1 shows a block diagram of a typical UVM testbench (VerificationGuide).

The different components of a typical UVM framework (Accellera (2015)):

- **Testbench** - Instantiates the Design under Test (DUT) module and the UVM Test class, and configures the connections between them.
- **Test** - It is the top-level UVM Component in the UVM Testbench. Typically performs three main functions: Instantiates the top-level environment, configures the environment and applies stimulus by invoking UVM Sequences through the environment to the DUT .
- **Environment** - It is a hierarchical component that groups together other verification component. Typical components that are usually instantiated inside the UVM Environment are UVM Agents, UVM Scoreboards, or even other UVM Environments. The top-level UVM Environment encapsulates all the verification components targeting the DUT.
- **Scoreboard** - Checks the behavior of a certain DUT. It usually receives transactions carrying inputs and outputs of the DUT through UVM Agents, runs the input transactions through some kind of a reference model (the predictor) to produce expected transactions, and then compares the expected output versus the actual output.

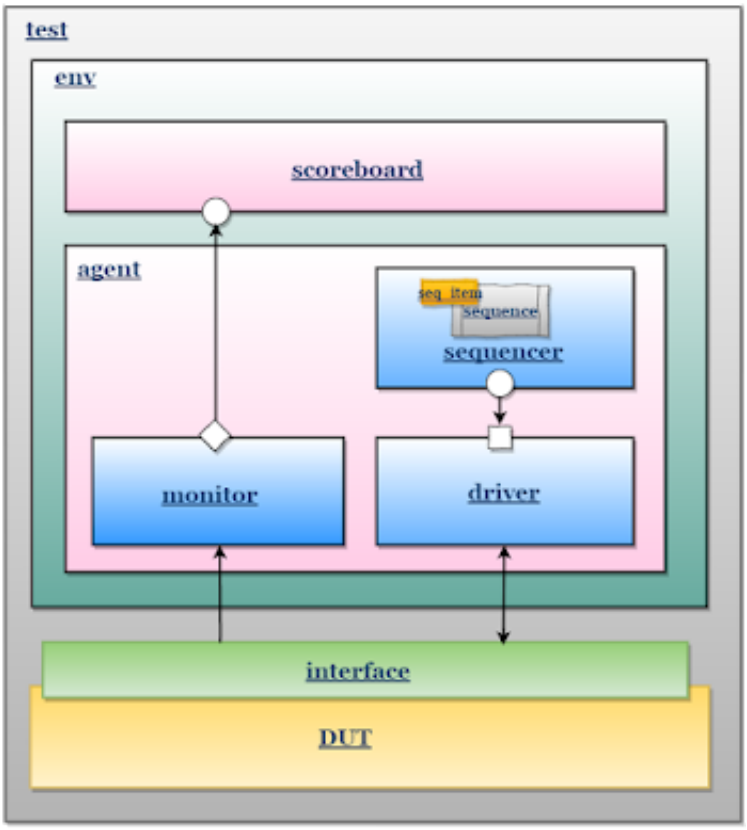

Figure 2.1: Block diagram showing different components of a typical UVM testbench

- **Sequencer** - It serves as an arbiter for controlling transaction flow from multiple stimulus sequences.
- **Driver** - It receives individual UVM Sequence Item transactions from the UVM Sequencer and applies (drives) it on the DUT Interface. Thus, it converts transaction-level stimulus into pin-level stimulus.
- **Monitor** - It samples the DUT interface and captures tthe input and output transactions and sends them to the rest of the testbench for analysis.
- **Agent** - It is a hierarchical component that groups together other verification components that deal with the DUT interface Typically it includes a Sequencer, a Driver and a Monitor.

In this project, the predictor (/model) is implemented in Python's PyMTL3 and the verification is carried out using the UVM framework implemented in Python Cocotb.

2.2 Model Development with PyMTL3

PyMTL3 is an open-source Python-based hardware generation, simulation, and verification framework with multi-level hardware modeling support (Jiang *et al.*, 2020 -). It provides a good number of libraries that facilitates model development at different levels: functional, RTL, Cycle Accurate. The reference model for this D-Cache verification project has been developed in PyMTL3. A detailed description of the model is provided in the next section.

2.3 Testbench Development with Cocotb

Cocotb is a COroutine based COsimulation TestBench environment for verifying VHDL and SystemVerilog RTL using Python (Higgs *et al.*, 2013 -). It uses a simulator to simulate the HDL design and can be used with a variety of simulators. For this project, Verilator has been used for simulating the RTL design. Though the UVM class library is developed in SystemVerilog, with cocotb, HDL is used for the design only, the test-bench is implemented in python. Thus it provides all the necessary UVM classes along with easier adaptability and smooth interfacing with the reference model as that is also python based.

CHAPTER 3

THE D-CACHE REFERENCE MODEL

3.1 General Design

The L1 data cache is a 8-way associative cache with a default size of 32KB and line size of 64B. It is non-blocking and other than normal data loads and stores, it also supports all fences and atomic instructions. It uses write-allocate policy where data is loaded to lines in the cache on store misses too. It writes-back to the main memory when dirty lines are replaced. The default replacement policy employed is pseudo-LRU (Least Recently Used).

The Load-Store Unit(LSU) is the only unit in the core that interfaces with the L1 Data Cache and consists of the Load Queue (LQ), the Store Queue (SQ) and the Fence and Atomic Buffer (FAB). The D-Cache is designed to receive requests of one of these three types :

- Store request at store commit time
- Load request from Load Buffer
- Load request from Store Buffer

The Load request from the Store Buffer is for prefetching the line in case of a miss so that the rest of the instructions are not blocked because the commit is stalled due to a store miss.(Shakti (2018-))

3.2 Model Implementation

The model is completely implemented in PyMTL3 and uses component classes in an hierarchical fashion. Emulating the RTL design, the model is associative and uses write-allocate with write-back policy. The model uses a Bit-PLRU (pseudo-LRU) replacement policy which is discussed in the next section. The model currently handles store commit requests, load requests from load buffer and all atomic operations. It is a parametric model with parameters : *naddr* (number of bits in the address), *nway* (assosciativity of the cache), *lineSize* (size of each cache line considering only data), *cacheSize* (size of cache considering only data) and *wordSize*; that are given as arguments during construction of the top-level module. The different component classes and the control flow of the design is given in the subsequent sections. Fig. 3.1 shows the block diagram of the D-Cache reference model. Note that since the model is functional the next level memory is also instantiated inside the model.

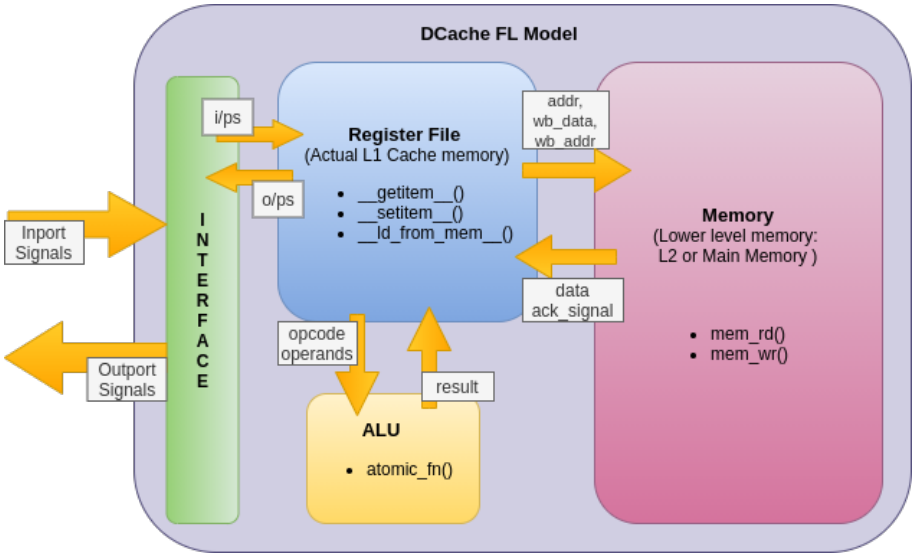

Figure 3.1: Block Diagram of D-Cache Reference Model

3.2.1 Bit-PLRU Replacement Policy

In the Bit-PLRU (pseudo Least Recently Used) replacement policy, one status bit is allotted to each cache line. These bits are called MRU (Most Recently Used) bits. A value of 1 on the MRU-bit indicates that the line has been accessed recently. Each

time a line is accessed its MRU-bit is set to 1. Whenever a set's last line with 0 on its MRU-bit is accesssed, its MRU-bit is set to 1 and those of all the other lines in the set are reset to 0. So at any point of time, there is atleast on line in every set whose MRU-bit is 0. During a cache miss, the first line in the set whose MRU-bit is 0 is replaced and its MRU-bit is set to 1. If the rest of the lines in the set have MRU-bit as 1, they are all reset to 0.

Since we use only one bit, the order in which the lines were accessed is not actually stored and hence we might end up replacing a line which was "more" recently accessed than another line in the set. Thus the algorithm's performance can take a hit but it requires less space, is fast and is easy to implement.

3.2.2 ALU

The class ALU defines the ALU component in the L1 cache used for Atomic operations. It takes *wordSize* as an argument at construction which is by default 4 (four bytes).
Functions:

1. **atomic_fn(opcode, loaded, rs2, data_width)**
   It carries out the atomic operation specified by the *opcode* on operands *loaded* (operand1) and *rs2* (operand2) and returns the result. The different operations include addition, bitwise-or, and, xor, signed and unsigned max, signed and unsigned min and returns operand2 by default.
   The *data_width* argument specifies whether the operation is on word or double. In case the operation is on word-sized operands, they may or may not be sign-extended depending on the opcode. The atomic operations follow a load-op-store sequence. The operand *loaded* is the data read from the memory address specified in the instruction and *rs2* is the data provided in the instruction, in accordance with the RISC-V spec.

3.2.3 Memory

The class Memory defines a memory component which could be used to instantiate the L2 cache or the main memory. It takes *mem_width*, the width of the memory block in bytes (which by default has been made equal to the lineSize of the L1 cache) and *naddr*, the number of bits in the address as arguments at construction. It has a register of type 'pymtl3.Bits' which is populated with data from the relevant files

during simulation.

Functions:

1. **mem_read(r_en, addr_in)**
   If *r_en* (read enable) is asserted, the function returns *'mem_width'* bytes of data from the memory register starting from the address specified by *addr_in*.

2. **mem_write(w_en, wb_data, wb_addr)**
   If *w_en* (write enable) is asserted, the function overwrites the data in the memory register with *wb_data* (write-back data) starting from the address specified by *wb_addr* (write-back address). Note that this function is called only when a dirty line is being replaced.

3.2.4 RegisterFile

The class RegisterFile defines a memory register which is instantiated in the L1 cache and functions as the cache memory. It takes the arguments *naddr* , *nway* , *lineSize*), *cacheSize* and *wordSize* at construction. It has a memory register which is an array of type 'pymtl3.Bits'. Apart from data, each element of the array contains bit fields for the **valid bit, dirty bit, tag** and the **MRU bit**. The class also has functions that carry out data movement operations along with book keeping and accepting responses from the next lower level memory block.

Functions:

1. **__getitem__(addr_in, data_width)**
   The *__getitem__* function takes the argument *addr_in* and computes the tag, index, and the byte offset and checks for the availability and validity of the requested data in the register. The argument *data_width* specifies the size of the data to be returned.

   We have the following possible scenarios:

   (a) CACHE HIT: If its a hit, it returns *is_hit* (asserted if and only if its a hit) and the requested data starting from the byte offset.

   (b) CACHE MISS:

   - Vacant Line Available: In this case it return *is_hit* (which will be de-asserted), *is_vacant* (asserted if and only if a vacant line is available), *vacantLine* (line address of the chosen vacant line).

   - Vacant Line Not Available:

- Replaced Line is Dirty: In this case it returns *is_hit*, *is_vacant*, *is_dirty* (asserted if and only if the chosen line is dirty), *repLine* (line address of the line to be replaced), *wb_addr* (write-back address), *wb_data* (write-back data).

- Replace Line is Not Dirty: The same set of variables as in the dirty line case are returned except that *is_dirty* is de-asserted and the *wb_addr* and the *wb_data* are set to zero and are irrelevant.

2. **__setitem__(addr_in, data_in, data_width)**
   The *__setitem__* function takes the argument *addr_in* and computes the tag, index, and the byte offset and checks for the availability and validity of the data in the requested location in the register. The argument *data_in* provides the data to be stored and *data_width* specifies the size of the data to be stored in the register.

   We have the following possible scenarios:

   (a) CACHE HIT: If its a hit, it overwrites the data in the register with *data_in* starting from the byte offset and returns *is_hit* (asserted if and only if its a hit).

   (b) CACHE MISS:

   - Vacant Line Available: In this case it return *is_hit* (which will be de-asserted), *is_vacant* (asserted if and only if a vacant line is available), *vacantLine* (line address of the chosen vacant line).

   - Vacant Line Not Available:

     - Replaced Line is Dirty: In this case it returns *is_hit*, *is_vacant*, *is_dirty* (asserted if and only if the chosen line is dirty), *repLine* (line address of the line to be replaced), *wb_addr* (write-back address), *wb_data* (write-back data).

     - Replace Line is Not Dirty: The same set of variables as in the dirty line case are returned except that *is_dirty* is de-asserted and the *wb_addr* and the *wb_data* are set to zero and are irrelevant.

3. **__ldFromMem__ (addr_in, data_in, data_width, op, targetLine, rd_resp_from_mem)**
   The *__ldFromMem__* function loads the data from memory *rd_resp_from_mem* to the line address in the register specified by the argument *targetLine* and sets the valid bit and the MRU bit, resets the dirty bit and updates the tag (computed from *addr_in*). Then, based on the value of the argument *op* (operation: load or store) it does one of the following:

   - Load - returns the requested data of size *'data_width'* from the freshly loaded line starting from the byte offset (computed from *addr_in*).

- Store - overwrites the freshly loaded line in the register with *data_in* of size *'data_width'* starting from the byte offset and sets the dirty bit to 1.

3.2.5 DCache

The class DCache derives from the Component class of PyMTL3 and is the top-level module that instantiates objects of the above described three classes namely, RegisterFile, Memory and ALU. It takes the arguments *naddr* , *nway* , *lineSize* , *cacheSize* and *wordSize* at construction and passes them onto the instantiated objects as applicable. It provides an interface to receive requests from and send out responses to the test bench. It methodically calls the different functions of the instantiated RegisterFile, Memory and ALU objects to carry out the received requests. Note that this is a functional level model and the Memory is instantiated here so that the model completes any request in one simulation clock cycle.

Interface:

Inports:

- req_from_core - (160 bit) request from core (added for verification/display purposes)
- addr_in - (64 bit) address for load, store and atomic operations
- op - (2 bit) specifies operation; 00 - Load, 11 - Store, 01 - Prefetch, 10 - PTW (01,10 are not suppported currently)
- atomic_op - (5 bit) opcode for atomic operations
- is_atomic - (1 bit) set if the request is an atomic operation
- data_in - (64 bit) data to be stored in the provided address in *'addr_in'*
- data_width - (2 bit) specifies the size of data to be loaded or stored; 00-Byte, 01-Half, 10-Word, 11-Double
- data_sign - (1 bit) specifies extension of the data to be read; 0-sign extended, 1-zero extended
- dest_type - (1 bit) specifies the type of the destination register; 0-Int, 1-Floating Point(FP)

Outports:

- ack - (1 bit) acknowledges the request; 1-accepted, 0-rejected

- done - (1 bit) shows task status; 1-completed, 0-in progress
- data_out - (64 bit) data sent as response to core for a load request

It also contains an internal 'rdy' bit which indicates that the cache is ready to accept a new request when it is High. If the cache is occupied, the 'rdy' bit is Low.

The DCache class contains the function *'operation'* which uses the *@update_ff* decorator and is called at every raising clock edge. It receives stimulus from the inports, carries out the request and updates the outports at the next rising clock edge. The sequence of steps carried out in this function for different types of requests are presented in the form of flow charts.
Once a request is received, the acknowledge bit is set high and the ready bit is set low and depending on whether its a load or store or atomic operation, it follows a sequence of steps. Fig. 3.2 shows the control flow for calling one of the operations: Load, Store or Atomic operation.

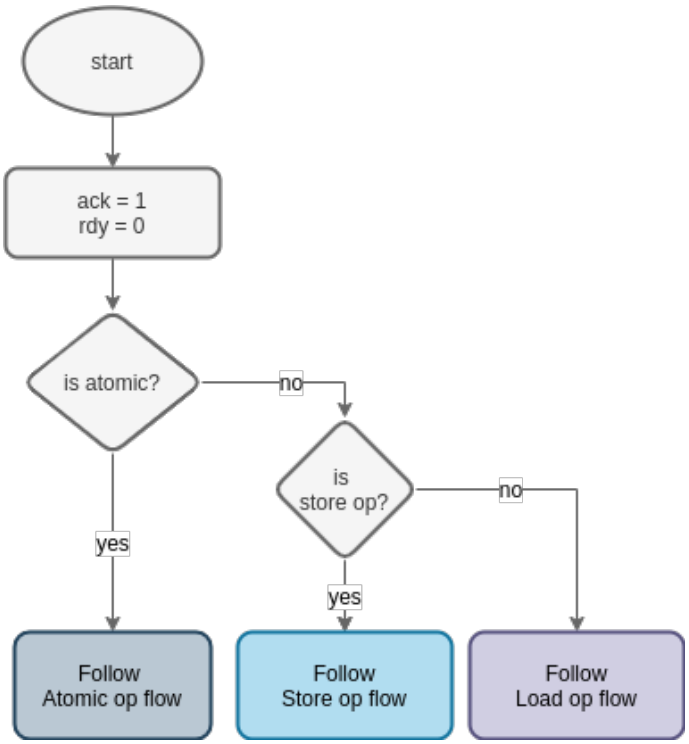

Figure 3.2: Flow chart showing control flow to select one of the operations.

For store operations, first the *__setitem__()* function is called and depending on whether its a hit or a miss; availability of vacant line in case of a miss and if the re-

placed line is dirty or not, in case no vacant lines are available, the functions *mem_rd()* and *mem_wr()* are called and once the operation is completed, the done signal is asserted. Fig. 3.3 shows the control flow for Store operations.

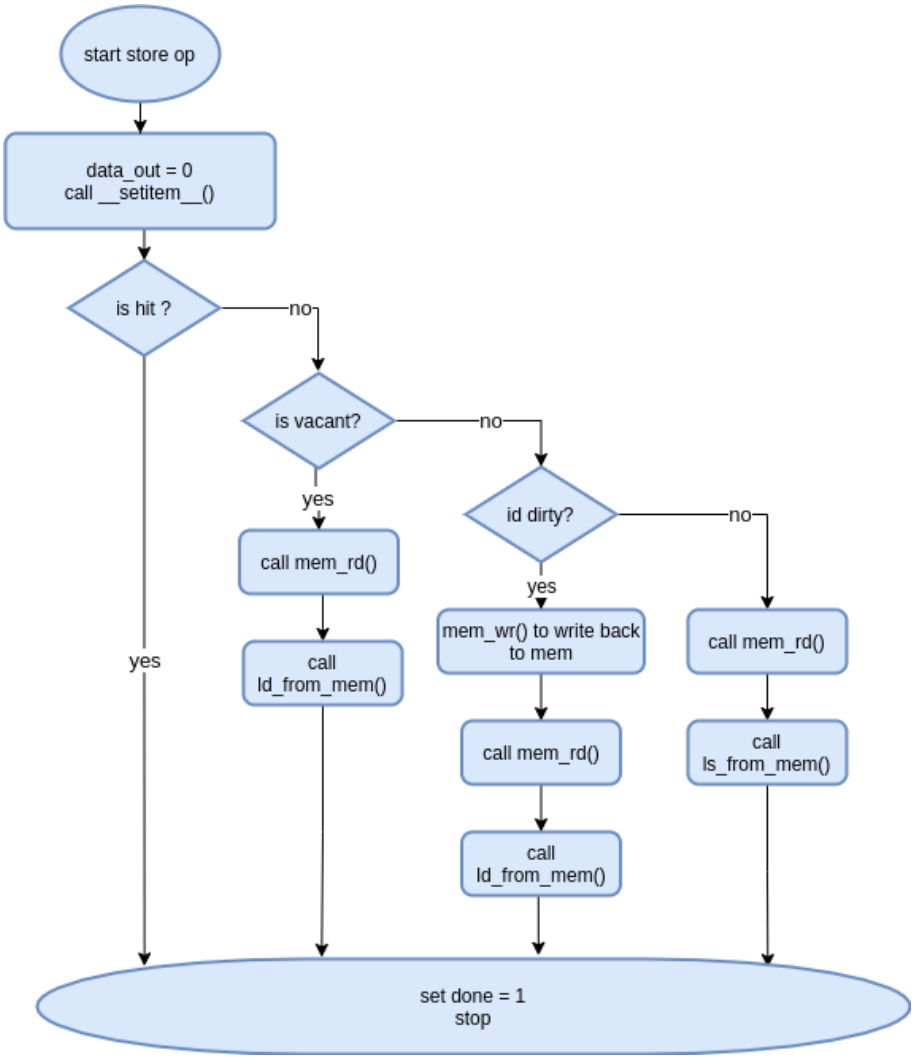

Figure 3.3: Control flow for store operations

Similar to store operations, for load operations, first the *__getitem__()* function is called and depending on whether its a hit or a miss, availability of vacant line in case of a miss and if the replaced line is dirty or not in case no vacant lines are available, the functions *mem_rd()* and *mem_wr()* are called. If the data is of type Int, depending on the extension specified in the request, it is either zero extended or sign extended to 64 bits. If its a float operation and data is not a double, we insert a string of 1s in the upper bits to make it 64bits wide (*NaN Boxing*). The resulting 64 bit data is made available at the data_out outport and the done signal is asserted. Fig. 3.4 shows the control flow for Load operations.

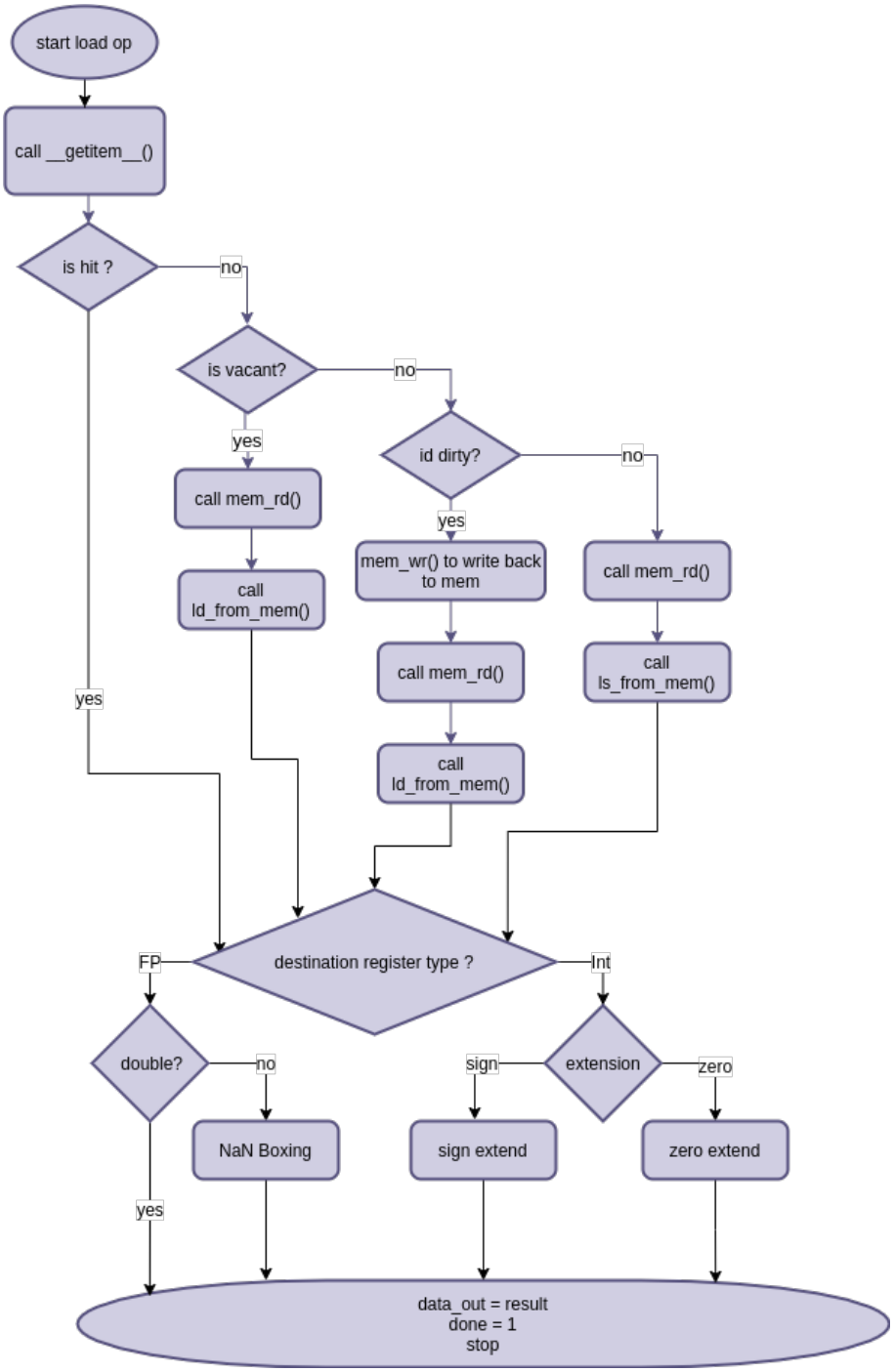

Figure 3.4: Control flow for load operations

Atomic operations follow the flow for loads till before extension. The loaded data, the data and the opcode specified in the request are given as arguments to *atomic_fn*. The result is then stored to the specified address following the control flow for stores. Fig. 3.5 shows the control flow for Atomic operations.

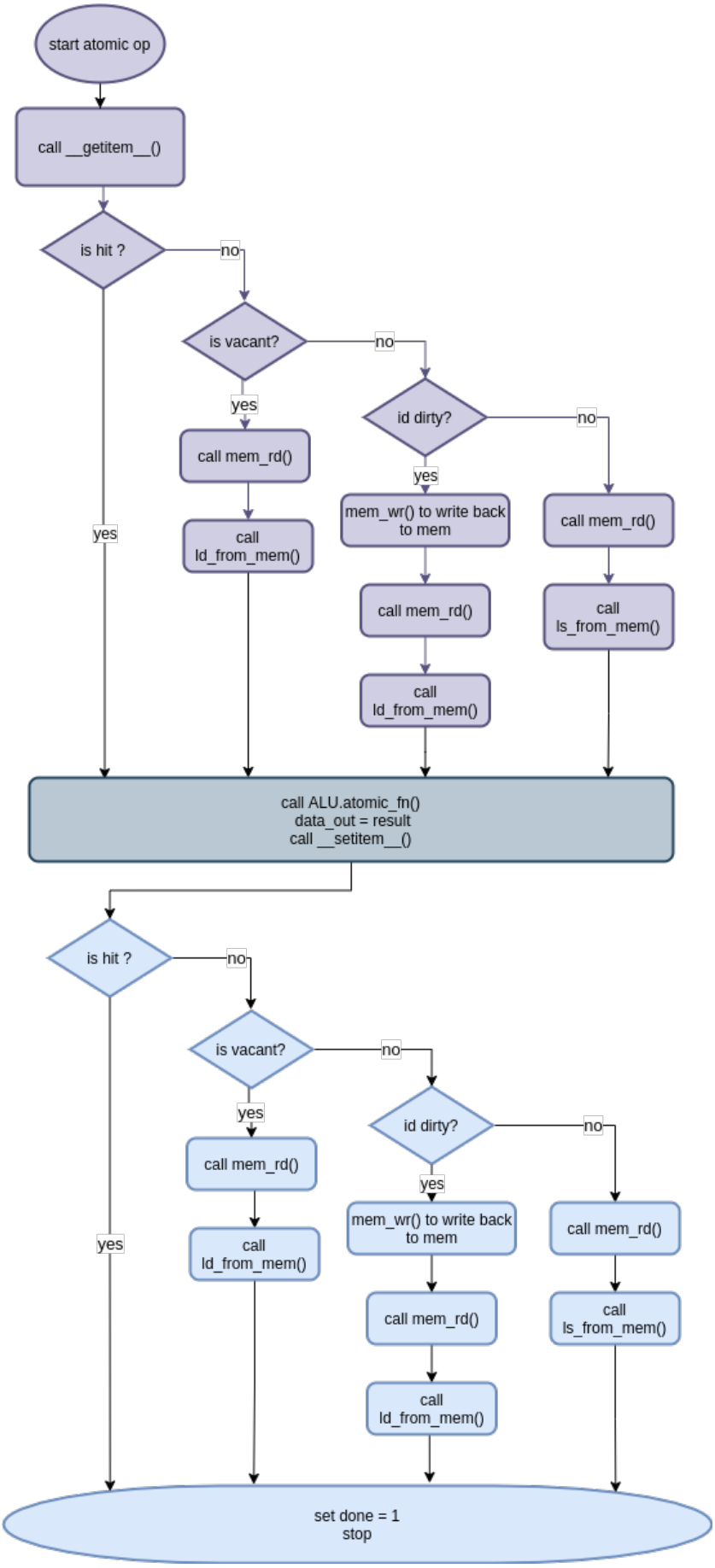

Figure 3.5: Control flow for atomic operations

CHAPTER 4

VERIFICATION

4.1 Steps in Verification

Steps followed in D-Cache Verification:

- Developing a verification plan
- Developing a test bench
- Simulation based verification
- Coverage analysis

This chapter provides the verification plan and explains the test bench developed for D-Cache verification and core level integration.

4.2 Verification Plan

The first step in verification is to develop a verification plan. It consists of identifying all the features, functionality or scenarios of interest and developing test strategies to exercise them to find potential bugs. Then based on this we can generate suitable tests to verify all the scenarios.
The verification plan for the D-Cache is given in the next page in the form of a table. The *Features* column contains the features and scenarios we want to test and the *Sub Features* column gives relevant finer details regarding the feature. The *Testplan* column contains the proposed test strategy to verify the feature and *Design Parameters* provides the relevant DUT signals to be monitored.

Table showing the verification plan for D-Cache.

TestPlan Id	Features	Sub Feature	Testplan	Design Parameters
dcache_0	**Cache memory register map with datafiles**	addr range: 0x1000-0x2000 : bootfile > 0x8000000 : code.mem	1. Feed load instructions accessing data from addresses from both the ranges.	subifc_req_from_core_put, subifc_resp_to_core_get
dcache_1	**Loads from Load Buffer**			subifc_req_from_core_put, subifc_resp_to_core_get
dcache_1_1	**Dest_type**	Specifies whether destination register type is float or int. Determines whether output should be NaN boxed or not. If data_width is less than 64 bits, all higher bits are made 1 (NaN Boxing)	1. Feed instructions covering both types of destination registers in combination with different data_widths.	subifc_req_from_core_put, subifc_resp_to_core_get
dcache_1_2	**Access Size**	Specifies the width of the data and its extension (zero or sign) to be returned. MSB: 0 - Sign ext; 1 - Zero ext LSBbits: 00 - Byte; 01 - Half; 10 - Word; 11 - Double	1. Feed instructions for all possible values of Access Size field	subifc_req_from_core_put, subifc_resp_to_core_get
dcache_2	**Stores from Store Commit**			subifc_req_from_core_put, subifc_resp_to_core_get
dcache_2_1	**Access size**	Specifies the width of the data and its extension (zero or sign) to be returned. MSB: 0 - Sign ext; 1 - Zero ext LSBbits: 00 - Byte; 01 - Half; 10 - Word; 11 - Double	1. Feed instructions for all possible values of Access Size field	subifc_req_from_core_put, subifc_resp_to_core_get
dcache_3	**Atomic Operations**	Follows load-compute-store Blocking operation Applicable only to 32 bit and 64 bit operands.		subifc_req_from_core_put, subifc_resp_to_core_get
dcache_3_2	**Opcode**	5 bit code specifying operation type and extension if any. Returns operand 2 in default case.	1. Feed instructions with all possible combinations of operations and extension.	subifc_req_from_core_put, subifc_resp_to_core_get
dcache_3_1	**is_atomic**	Specifies if the operation is atomic or not. 0 - Not atomic; 1 - atomic	1. Monitor the signal and see if atomic operations are carried out whenever the bit goes high.	subifc_req_from_core_put, subifc_resp_to_core_get
dcache_3_3	**loaded (operand 1)**	Data loaded from the specified address used as operand 1.		subifc_req_from_core_put, subifc_resp_to_core_get
dcache_3_4	**rs2 (operand 2)**	Data specified in the instruction used as operand 2.		subifc_req_from_core_put, subifc_resp_to_core_get
dcache_3.5	**Access size**	Specifies the width of the data and its extension (zero or sign) to be returned. MSB: 0 - Sign ext; 1 - Zero ext LSBbits: 00 - Byte; 01 - Half; 10 - Word; 11 - Double	1. Feed instructions for all possible values of Access Size field	subifc_req_from_core_put, subifc_resp_to_core_get
dcache_4	**Write back**	When dirty line is replaced, data is written back to the lower level memory.	1. Carry out a store operation. 2.Consecutive accesses to address locations mapping to same cache line. 3. Carry out Load operation from the earlier address and compare data.	subifc_req_from_core_put, subifc_resp_to_core_get
dcache_5	**Load after Store**		1. Carry out a store operation. 2. Carry out Load operation from the earlier address and compare data.	subifc_req_from_core_put, subifc_resp_to_core_get
dcache_6	**Load afterAtomic op**		1. Carry out an atomic operation. 2. Carry out Load operation from the earlier address and compare data. Repeat for all combinations of opcodes and data widths.	subifc_req_from_core_put, subifc_resp_to_core_get

4.3 Testbench Implementation

The verification setup is implemented entirely in Python cocotb. The functionality of different components essentially remains the same as discussed in chapter 2. A brief description of the components along with the conditions they check is given in the subsequent sections. Fig. 4.1 shows the block diagram of the verification setup.

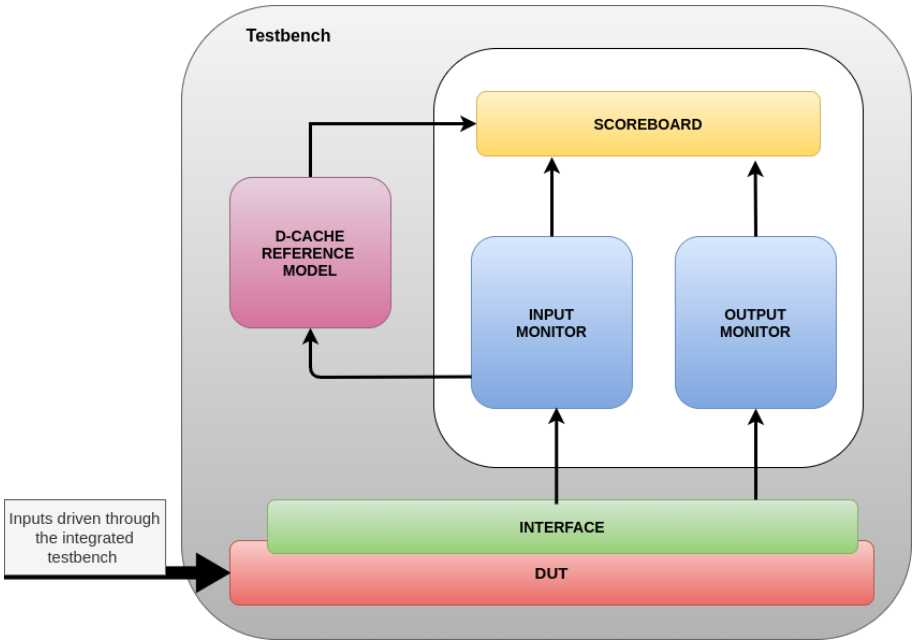

Figure 4.1: Block Diagram of the verification setup

4.4 Components

4.4.1 Input Monitor

It is derived from the cocotb class *'BusMonitor'* and samples input signals received by the DUT based on the conditions that the signals *'EN_subifc_req_from_core_put'* (enable) and *'RDY_subifc_req_from_core_put'* (ready) are High. Every time it captures an input transaction, it calls back to the reference model and passes on the input transaction.

4.4.2 Output Monitor

It is derived from the cocotb class *'BusMonitor'* and samples output signals sent out by the DUT based on the conditions that the signals *'EN_subifc_resp_to_core_get'* (enable) and *'RDY_subifc_resp_to_core_get'* (ready) are High. Every time it captures an output transaction, the Scoreboard is called.

4.4.3 Scoreboard

It is derived from the cocotb class *'Scoreboard'*. It is interfaced with the Output Monitor and the expected result array of the reference model. Once the Scoreboard receives a new transaction from the Output Monitor, it searches the expected result array for a response corresponding to the request with ROB (reorder buffer) ID matching the ROB ID of the received output transaction. This is necessary because the core supports out-of-order-execution. Then it checks for any exceptions (like load access fault, page faults etc.). If there are no exceptions, the output from DUT is compared with the expected output. To check for mismatches in the case of loads, the returned data is compared and for atomics, the stored result is compared. For store instructions, it only checks if a response with a certain ROB ID has been received or not.

4.4.4 Testbench

It is the top-level model that instantiates the components (input and output monitors, scoreboard and the reference model) and connects them with each other. It interfaces with the reference model through a wrapper function to feed the model with the input transaction captured by the input monitor.

4.5 Block Level Verification and Core Level Integration

Block level verification does give us a greater control over the signals driven into the DUT and the model. But the cache module needs to interact with the memory, so it no longer remains a single block for simulation. We moved on to core level integrated testbench as it is much easier to implement and facilitates verification of the entire I-class core in future. To retain the control over signals, we can generate tailored tests to suit the proposed test plan with a larger number of memory instructions and special patterns to further check any corner cases.

The core level test bench is modified to especially suit for D-Cache Verification. It instantiates the D-Cache test bench shown in Fig. 4.1 and passes to it the handle to D-Cache of the I-class core DUT. The block diagram of the complete verification setup is shown in Fig. 4.2.

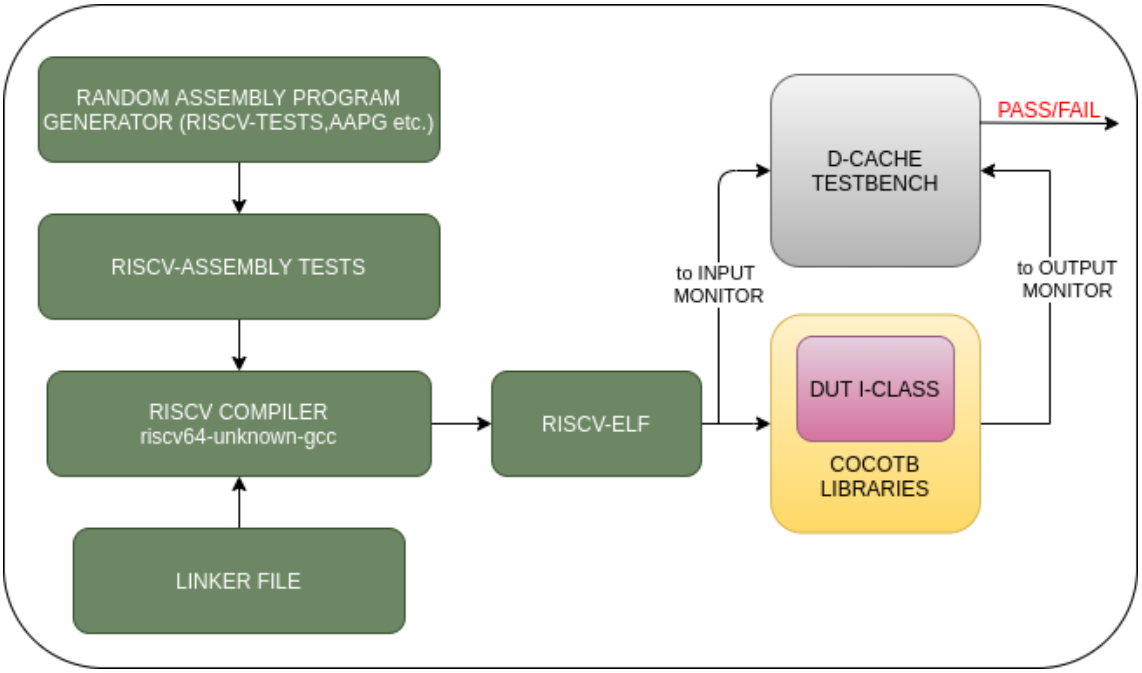

Figure 4.2: Block Diagram of the verification setup

CHAPTER 5

RESULTS

Coverage is defined as a measure of how much of the DUT functionality has actually been exercised by the application of tests. It is necessary that we cover all kinds of different scenarios through our tests to ensure that there are absolutely no bugs in the system. Coverage is a handy tool that allows us to identify which scenarios have been left out and generate tailored tests to activate them. Thus it is important to define a proper coverage plan which considers all the relevant signals and instruction patterns and ensures that each one of them takes all the possible values.

5.1 Coverage Plan

The coverage plan for D-Cache verification is shown in table 5.1.

Table 5.1: Table showing the coverage plan for D-Cache Verificaiton

S.no.	Coverpoint/patterns	Values
1	ext	sign, zero
2	access_size	byte, half, word, double
3	origin	load buff, store commit
4	dest_type	FP, int
5	is_atomic	Set, Unset
6	atomic_fn	op1,op1_s,op2,add,add_s, or,or_s,xor,xor_s,and,and_s, minu_s,maxu_s,maxu,minu, max_s,min_s,max,min
7	RAW	Set, Unset
8	RAA	Set, Unset
9	en_req_frm_core	Set, Unset
10	en_flush	Set, Unset
11	en_rd_req_to_mem	Set, Unset
12	en_wr_req_to_mem	Set, Unset

RAW-load after store to same address; RAA-load after atomic op to same address

5.2 Test Suite

Various tests covering different types of scenarios have been used to carry out the proposed verificaion plan. The tests used are of two types based on how they are generated: RISCV and AAPG.

5.2.1 RISCV Tests

RISCV tests are based on hard coded assembly programs with a fixed number and types of instructions. The aim is to verify the basic functionality of the DUT by feeding instructions that request different operations. We have a large number of these RISCV tests each of which specializes in a specific functionality. For instance we have:

- *amoor_d* - targets atomic OR function on *double* sized operands.
- *amoor_w* - targets atomic OR function on *word* sized operands.
- *amominu_d* - targets atomic function returning minimum of the two unsigned *double* sized operands.
- *ammin_w* - targets atomic function returning minimum of the two signed *word* sized operands.
- *lw* - targets load word functionality.

Tools used: *riscv-gnu-toolchain* and *riscv-isa-sim*.

The 'spike.dump', 'rtl.dump' and 'disassembly' files are automatically generated and added to the test directory during test generation.

5.2.2 AAPG

AAPG stands for Automated Assembly Program Generator. It is a Python based assembly program generation tool developed by Shakti. It allows the programmer to configure the number of instructions, the ratio of different types of instructions and any customized patterns of instructions, like the repeated store-load or atomic-load combinations to the same address used for this project. It uses these configurations as constraints and generates random assembly tests. This is an immensely useful tool as

it allows a certain level of control and yet randomizes the instructions thus enabling us to target a particular functionality with tailored instruction patterns and exhaustively check for any potential bugs.

Tools used: *AAPG*, *riscv-gnu-toolchain* and *riscv-isa-sim*.
Configurations are added in 'config.yaml' file. Similar to RISCV tests, log and dump files are also generated.

5.3 COVERAGE

5.3.1 Functional Coverage

The functional coverage data obtained from running all the tests have been merged to obtain the overall functional coverage figures. Fig. 5.1 shows the merged functional coverge.

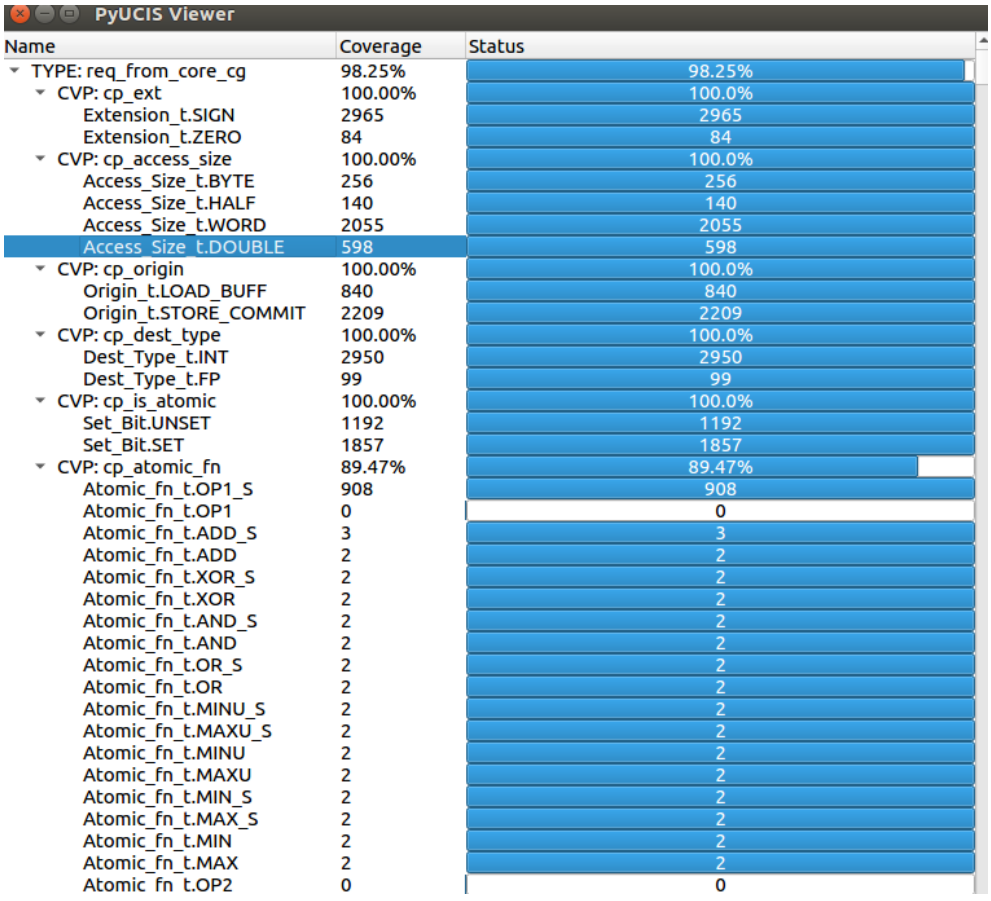

Figure 5.1: Functional coverage statistics for different fields of the instruction from pyucis-viewer

CHAPTER 6

FUTURE WORK

- The model can be further developed to support prefetching and PTW requests.
- More complex coverage points can be added after adding more functionality to the model.
- The current model is a functional level model and verifies functional correctness only. The model can be made cycle accurate to evaluate performance and timing.

REFERENCES

1. **Accellera** (2015). Universal verification methodology (uvm) 1.2 user's guide. URL https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf.

2. **Higgs, C.** *et al.* (2013 -). Cocotb documentation. URL https://docs.cocotb.org/en/stable/.

3. **Jiang, S.** *et al.* (2020 -). Pymtl3 framework. URL https://pymtl3.readthedocs.io/en/latest/.

4. **Shakti** (2018-). Caches_mmu gitlab repository. URL https://gitlab.com/shaktiproject/uncore/caches_mmu.

5. **Shakti** (2019 -). Aapg. URL https://gitlab.com/shaktiproject/tools/aapg/.

6. **VerificationGuide** (). Uvm testbench. URL https://verificationguide.com/uvm/uvm-testbench/.