جامعة القاهرة

Functional Verification of OpenHMC

Submitted By

Alaa Rabie Salem

Mohamed Ahmed Taha

Ali Maher Abdelsalam

Mohamed Hosam Othman

Mahmoud Mousa Gad

Nada Othman Mohamed

Rahma Hamdy Abdullah

Graduation Project Thesis

Submitted to Faculty of Engineering, Cairo University in Partial Fulfilment of the Requirements for the Degree of Bachelor of Science in Electronics & Communications Engineering

Supervisor

Prof. Ibrahim Qamar

July 2023

Sponsor

Si-Vision

جامعة القاهرة	Academic Year:	2022/2023	Term:	Spring 2023	FOUNDED IN 1816
	Course Code:	ELC 4802	Course Title:	Project (2)	

Cairo University
Faculty of Engineering
Electronics and Communications Engineering Department
4th Year

Graduation Project
Functional Verification of OpenHMC

- Sponsored by Si-Vision -

Submitted By

Alaa Rabie Salem - 9191445

Ali Maher Abdelsalam - 9190067

Mahmoud Mousa Gad - 9190971

Mohamed Ahmed Taha - 9191043

Mohamed Hosam Othman - 9191083

Nada Othman Mohamed - 9191568

Rahma Hamdy Abdullah - 9190636

Supervised By

Prof. Ibrahim Qamar

ACKNOWLEDGEMENTS

We are using this opportunity to express our gratitude to everyone who supported us throughout the graduation project.

Firstly, we would like to express our sincere gratitude to our supervisor **Dr. Ibrahim Qamar** for the continuous support and guidance.

We also wish to thank our advisors at **Si-Vision**, **Eng. Ahmed Abdelbaky**, **Eng. Ahmed Abdelrahman**, **Eng. Hagar Fathy**, **Eng. Mohamed Younis**, **Eng. Mostafa Sarhan** and **Eng. Yahia Zakaria** for their patience, motivation, and immense knowledge.

Finally, we wish express the gratitude and appreciation we owe to our families for providing us with unfailing support and continuous encouragement throughout our years of study.

This accomplishment would not have been possible without all of you. Thank you.

ABSTRACT

Digital designs are getting more complex day by day, which makes testing them using basic testbench insufficient anymore. Because of this, digital verification has a major role in industry. Digital verification builds confidence in the design and makes sure that it accomplishes its tasks successfully. An optimum way to do so is by adopting constrained random stimulus and coverage-driven verification. There are many methodologies that are used over the years for verification, the latest is The Universal Verification Methodology (UVM). It is a complete methodology that codifies the best practices for efficient and exhaustive verification.

In this book, we propose our verification plan and UVM environment for the functional verification of OpenHMC controller. The detailed implementation of our environment, the testcases to verify the OpenHMC and simulation results are discussed in the book as well.

TABLE OF CONTENTS

Chapter 1 Introduction 1
1.1 Functional Verification 1
1.1.1 ASIC Digital Design Flow 1
1.1.2 The Goal of Functional Verification 2
1.1.3 The Features of SystemVerilog 2
1.2 The Features of UVM 2
1.3 Goals of the project 3
1.4 Organization of the book 4
Chapter 2 Introduction About RTL 5
2.1 What's OpenHMC? 5
2.2 Features 5
2.2.1 Supported Configurations 6
2.3 Module Description 6
2.3.1 Top Module (openhmc_top.v) 6
2.3.2 Transmit and Receive FIFOs 7
2.3.3 TX Link (tx_link.v) 7
2.3.4 RX Link (rx_link.v) 12
2.3.5 Register File (openhmc_rf.v) 13
2.3.6 Header File 13
Chapter 3 HMC Architecture & Specs 14
3.1 Interfaces, pins and configuration parameters 14
3.1.1 Configuration parameters 14
3.1.2 Interfaces and Pins 15
3.2 Register File contents and its address map 21
3.3 TS1 (Training Sequence) Definition 26
3.4 Lane Run Length Limitation 27
3.5 Lane Reversal 28
3.6 Lane Polarity 28
3.7 Link Layer 28
3.8 Transaction Layer 29

3.8.1 Packet Length 31
3.8.2 Packet Flow Control 31
3.8.3 Tagging 32
3.8.4 Packet Integrity 32
3.8.5 Request Packets 33
3.8.6 Response Packets 35
3.8.7 Flow Packets 40
3.8.8 Request Commands 40
3.8.9 Response Commands 46
3.8.10 Flow Commands 48
3.8.10.3 TOKEN RETURN (TRET) Command 49
3.8.11 Valid Field Command Summary 49
3.8.12 Transaction Layer Initialization 49
Chapter 4 Verification Plan & Features 51
4.1 Design Requirements Sheet 51
4.2 System Sheet 52
4.3 Interfaces Sheet 52
4.4 Generation Sheet 53
4.5 Checking Sheet 53
4.6 Coverage Sheet 54
4.7 Test List Sheet 54
4.8 RF Map Sheet 55
Chapter 5 Testbench Architecture 56
5.1 Main Code Files 56
5.1.1 tb_top module 56
5.1.2 Packages 57
5.1.3 Environment 57
5.1.4 Configuration Class 57
5.1.5 System Interface 57
5.1.6 hmc_pkt_item class 57
5.1.7 Other Interfaces 57
Chapter 6 RF Agent 59

6.1 Register File Module 59
6.2 Register File Interface 59
6.3 Bad Register File Contents Documenting 60
6.4 RF Agent Implementation 63
6.4.1 Registers 64
6.4.2 Register Block 65
6.4.3 Adapter 65
6.5 RF Interface Assertions 66
6.6 RF Sequences 67
6.6.1 Initialization Sequence 67
6.6.2 Sleep Sequence 69
6.6.3 Wrong Operation Sequence 72
6.6.4 Read and Reset Sequences 74
Chapter 7 AXI Agent 75
7.1 The AXI agent architecture 75
7.2 AXI agent components and other implemented objects 75
7.2.1 AXI interface 75
7.2.2 hmc_pkt_item_request 77
7.2.3 valid_data 77
7.2.4 Sequence 77
7.2.5 Sequencer 78
7.2.6 Config 78
7.2.7 Driver 78
7.2.8 Monitor 78
Chapter 8 HMC Agent 80
8.1 Initialization Stage 80
8.1.1 driver_hmc_agent 80
8.1.2 monitor_hmc_agent 82
8.1.3 Sequences 82
8.2 Request / Response Stage 83
8.2.1 link_up function 83
8.2.2 get_packets function 84

8.2.3 get_response function 85
Chapter 9 Scoreboard & Coverage Collector 86
9.1 Scoreboard 86
9.1.1 Predictor / Reference Model 86
9.1.2 Comparator 86
9.1.3 Connection between scoreboard, predictor and comparator 86
9.1.4 UVM Scoreboard types 87
9.1.5 Scoreboard Functions 88
9.1.6 report phase 89
9.2 Coverage Collector 89
9.2.1 Code Coverage 89
9.2.2 Functional coverage 91
Chapter 10 Codes Summary 94
10.1 Files ordering 94
10.1.1 Simulation Folder 95
10.1.2 Testbench Folder 96
10.2 Statistics 100
10.3 Contents of Major Files 102
10.3.1 Top Level 102
10.3.2 Base Test 103
10.3.3 Environment 105
10.3.4 Agent 106
10.3.5 Parameters Package 108
Chapter 11 Simulation and Results 109
11.1 Tools 109
11.2 Scripts 109
11.2.1 dut and tb files 110
11.2.2 waves file 112
11.2.3 Run file 112
11.3 Tests 115
11.3.1 Simple Test 116
11.3.2 Simple Test Simulation 118

11.3.3 Simple Test Results 123
11.3.4 Add tests with more constraints 128
11.3.5 Reset Tests 129
11.4 Results 130
11.4.1 Features and Configurations 130
11.4.2 Functional Coverage 131
11.4.3 Code Coverage 132
11.4.4 Finite State Machine Coverage 133
11.4.5 Assertions 135
Chapter 12 Conclusion 136
12.1 Points Learned 136
12.2 Problems Faced 136
12.3 Advices and Future Work 137
References 138

LIST OF FIGURES

Figure 1.1: ASIC Design Flowchart 1
Figure 1.2: main uvm classes 3
Figure 2.1: OpenHMC controller block diagram 5
Figure 2.2: Link Data Transmission Implementation Example 7
Figure 2.3: The Design Finite State Machine (FSM) 8
Figure 2.4: TX link diagram 9
Figure 2.5: Data-Reordering: 4FLIT/512bit example 9
Figure 2.6: RX link diagram 12
Figure 3.1: OpenHMC controller architecture in details 14
Figure 3.2: AXI-4 interface diagram 16
Figure 3.3: Example transactions on the AXI TX TDATA bus for FPW=4 17
Figure 3.4: TUSER Example for FPW=4 18
Figure 3.5: HMC physical layer (transceiver) interface diagrams with pins 18
Figure 3.6: System interface diagram with its pins 19
Figure 3.7: HMC interface diagram pins 20
Figure 3.8: Register file interface diagram with its signals 20
Figure 3.9: Register File Access: Write and read register 0x2 21
Figure 3.10: Packet layouts 29
Figure 3.11: Request Packet Header Layout 33
Figure 3.12: Request Packet Tail Layout 34
Figure 3.13: Response Packet Header Layout 35
Figure 3.14: Response Packet Tail Layout 36
Figure 4.1: Design Requirements Sheet 51
Figure 4.2: System Sheet 52
Figure 4.3: Interfaces Sheet 52
Figure 4.4: Generation Sheet 53
Figure 4.5: Checking Sheet 53
Figure 4.6: Coverage Sheet 54
Figure 4.7: test list Sheet 54
Figure 4.8: rf_map Sheet 55
Figure 5.1: Testbench Architecture 56
Figure 6.1: register file interface diagram 59
Figure 6.2: register file access, read and write operation 60
Figure 6.3: Control register RTL code 62
Figure 6.4: invalid address pin behavior when reading from counter reset register 62
Figure 6.5: Counter reset register RTL code 63
Figure 6.6: diagram of uvm register model and agent 63
Figure 6.7: UVM base classes for registers 64
Figure 6.8: a snapshot of status init register class 64

Figure 6.9: snapshot of uvm_reg_block class 65
Figure 6.10: register adaptor base class 65
Figure 6.11: Register File Assertions codes 66
Figure 6.12: first step in initialization sequence 67
Figure 6.13: waiting for phy_tx_ready in initialization sequence 68
Figure 6.14: status register field values during initialization and training 68
Figure 6.15: status general content after link is up 69
Figure 6.16: hmc monitor reporting during initialization 69
Figure 6.17: sleep scenario inside the DUT fsm 70
Figure 6.18: control register in sleep sequence 70
Figure 6.19: LxRXPS is driven to low during sleep 71
Figure 6.20: the hmc agent drives LxTXPS to low 71
Figure 6.21: status register content during sleep 71
Figure 6.22: exiting sleep mode 72
Figure 6.23: code of the register file wrong op seq 73
Figure 6.24: the response of the DUT invalid address during performing wrong operations 73
Figure 6.25: assertions related to rf wrong op seq 73
Figure 6.26: example of a reset sequence 74
Figure 7.1: AXI Architecture 75
Figure 8.1: Tx and RX state of the DUT 80
Figure 8.2: TS1 Ordering at every positive edge of the clock cycle 82
Figure 8.3: link_up function in hmc_agent_base_driver 83
Figure 8.4: get_packets function in hmc_agent_base_driver 84
Figure 8.5: get_response function in hmc_agent_base_driver 85
Figure 9.1: Connection between scoreboard, predictor, and comparator 87
Figure 9.2: Statement Coverage Example 90
Figure 9.3: Expression Coverage Example 90
Figure 9.4: State/Transition Coverage Example 91
Figure 9.5: Coverage Comparison 92
Figure 10.1: Github repo - Highest Level 95
Figure 10.2: github simulation folder 95
Figure 10.3: github coverage folder 96
Figure 10.4: github testbench folder 97
Figure 10.5: axi package file from inside 97
Figure 10.6: github axi agent folder 98
Figure 10.7: github register block folder 98
Figure 10.8: github sequence library folder 99
Figure 10.9: sequence and test packages 99
Figure 10.10: github test library folder 100
Figure 10.11: summary of lines of codes 100
Figure 10.12: github code frequency 101
Figure 10.13: github contributions 101

Figure 10.14: testbench top and instantiation of interfaces 102
Figure 10.15: how to instantiate the DUT 103
Figure 10.16: tb_top set the interfaces into uvm config db 103
Figure 10.17: base test file 104
Figure 10.18: base test build phase 104
Figure 10.19: environment class layout 105
Figure 10.20: environment build phase 105
Figure 10.21: environment connect phase 106
Figure 10.22: register file agent file layout 107
Figure 10.23: build and connect phase functions of rf agent 107
Figure 10.24: content of parameters package 108
Figure 11.1: used tools 109
Figure 11.2: scripts directory 110
Figure 11.3: dut script file 111
Figure 11.4: tb script file 111
Figure 11.5: waves script 112
Figure 11.6: first part of run file 113
Figure 11.7: snapshot from user manual of code coverage types 114
Figure 11.8: second snapshot of run file 114
Figure 11.9: last part of run file 115
Figure 11.10: simple test file 116
Figure 11.11: simple test virtual sequence 117
Figure 11.12: fsm during simple test 117
Figure 11.13: rf check vseq 118
Figure 11.14: snapshot of printing the testbench architecture 119
Figure 11.15: printing configuration object 119
Figure 11.16: configure the dut after starting test 120
Figure 11.17: monitor printing states 120
Figure 11.18: last part of states during initialization 121
Figure 11.19: axi printing the created packets 121
Figure 11.20: axi waves during axi sequence 122
Figure 11.21: sleep sequences printed messages during simple test 122
Figure 11.22: register counters of packets 123
Figure 11.23: initial functional coverage from simple test 124
Figure 11.24: shape of initial code coverage after simple test 124
Figure 11.25: initial coverage report from simple test 125
Figure 11.26: timetable of constrained random testing 125
Figure 11.27: FSMs of the DUT 125
Figure 11.28: state FSM coverage from simple test 126
Figure 11.29: init state machines after simple test 127
Figure 11.30: variable that gets randomized in axi sequence 128
Figure 11.31: show how randomization inside axi seq is based on the randomized variable 128

Figure 11.32: extending from axi base sequence and re write constraints 129
Figure 11.33: activate reset in base sequence 129
Figure 11.34: summary of the three covergroups 131
Figure 11.35: messing coverage commands 132
Figure 11.36: final code coverage summary 132
Figure 11.37: code coverage of openHMC top 132
Figure 11.38: code coverage of openHMC rf 133
Figure 11.39: rx and tx links code coverage 133
Figure 11.40: final state FSM coverage 134
Figure 11.41: initialization states fully covered 135
Figure 11.42: all concurrent assertions results 135

LIST OF TABLES

Table 2.1: TX FSM State Table 9
Table 2.2: TX FSM Transition Table 10
Table 2.3: RAM Configurations 11
Table 3.1: Configuration parameters in top module 15
Table 3.2: HMC physical layer (transceiver) interface signals 19
Table 3.3: Register file interface signals 21
Table 3.4: Register file address map 22
Table 3.5: status_general register 23
Table 3.6: status_init register 24
Table 3.7: Other Counters (Each Entry equals one Register) 25
Table 3.8: Performance Counters (Each Entry equals one Register) 25
Table 3.9: Control register 26
Table 3.10: TS1 bit position 27
Table 3.11: TS1 (Training Sequence) Example 27
Table 3.12: Request Packet Header Fields 34
Table 3.13: Request Packet Tail Fields 35
Table 3.14: Response Packet Header Fields 36
Table 3.15: Response Packet Tail Fields 37
Table 3.16: ERRSTAT[6:0] Bit Definitions 38
Table 3.17: Transaction Layer – Request Commands 41
Table 3.18: Operands from DRAM 43
Table 3.19: Immediate Operands Included in Atomic Request Data Payload 44
Table 3.20: Operand from DRAM 44
Table 3.21: Immediate Operand Included in Atomic Request Data Payload 44
Table 3.22: Mode Request Addressing 45
Table 3.23: Valid Data Bytes 45
Table 3.24: Request Packet Bytes to be written 46
Table 3.25: Transaction Layer – Response Commands 46
Table 3.26: Flow Commands 48
Table 3.27: Valid Field and Command Summary Table 49
Table 5.1: Register File Address Map 60
Table 6.2: Control register documentation of OpenHMC 61
Table 6.3: counter_reset documentation in openhmc 62
Table 6.4: Register File Assertions Explanation 66
Table 7.1: AXI interface signals description 76
Table 11.1: covered features in tests 130
Table 11.2: openhmc tested configurations 131

LIST OF ABBREVIATIONS

ADRS	Address
AXI-4	Advanced eXtensible Interface
CAG	Computer Architecture Group
CDR	Clock-Data Recovery
CMD	Command
CRC	cyclic redundancy check
CTRL	Control
DEMUX	De-Multiplexer
DINV	Data Invalid
DLN	Duplicate Length
DRAM	Dynamic RAM
DUT	Design Under Test
DWIDTH	Datapath
ERRSTAT	Error Status
FIFO	First In First Out
FPW	FLIT Per Width
FRP	Forward Retry Pointer
FSM	Finite State Machine
HMC	Hybrid Memory Cube
INIT	Initialization
IRTRY	Initialization Retry
LFSR	Linear Feedback Shift Register
LNG	Packet Length
LNK	Link
LSB	Least Significant Bit
LUTs	Look-Up Tables
OOP	Object Oriented Programming

PLLs	Phase-Locked Loops
PRET	Retry Pointer Return
RAL	Register Abstraction Layer
RAM	Random Access Memory
RES	Reserved
RF	Register File
RRP	Return Retry Pointer
RTC	Return Token Count
RTL	Register Transfer Level
RTRY	Retry
RX	Receiver
SEQ	Sequence Number
SLID	Source Link ID
Specs	Specifications
SVA	SystemVerilog Assertions
TCL	Tool Command Language
TGA	Return TAG
TRET	Token Return
TS1	Training Sequence
TX	Transmitter
UI	Unit Interval
UVM	Universal Verification Methodology

Chapter 1
Introduction

In this chapter, we will present some of the pre-requisites needed to know before diving into the project. As well as the goal of the project and organization of the book

1.1 Functional Verification

We need to understand the role and position of functional verification in the cycle of digital IC design flow. In addition, understand why the HDL language "Verilog" isn't enough for current requirements of verification of complex designs and why people moved to a new hardware verification language called "SystemVerilog" to build testbenches.

1.1.1 ASIC Digital Design Flow

It can be best explained by a flowchart of the typical steps needed for the fabrication of an IC as shown in (Figure 1.1).

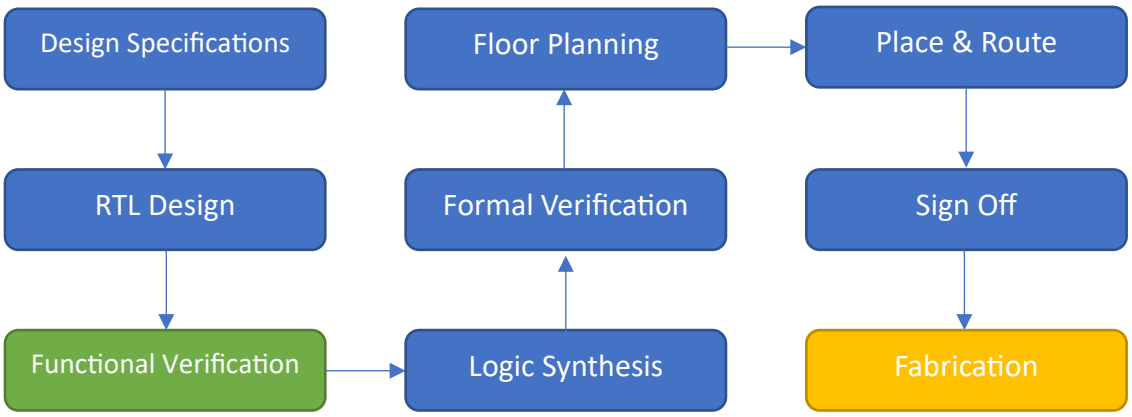


Figure 1.1: ASIC Design Flowchart

Typically, in industry field. The chip functionality is described in "Requirement & Specification Document" by the system engineer according to agreement with the customer. The Design Engineer starts to plan the architecture blocks of the design to meet the system requirements. Then model it using a hardware description language like Verilog/VHDL but currently Verilog is the most commonly used in the industry. This language has to be synthesizable as the code must be compatible with the synthesis tool to represent logic gates. In parallel with the design process of the RTL, The Verification Engineer builds the verification environment to test the functionality of the design. To make sure the RTL meets all the specs. By simulating multiple test cases and reporting functional coverage once the first version of the RTL is ready. If either a bug is found or the design not achieving a spec then report back to design engineer.

Discovering any bugs at this early stage before heading into the synthesis and the backend implementation of the IC is very benefitable for multiple reasons, some of them are:

- Easier to find the error and fix the bug
- Possibly saving time and money of repeating the flow of fabrication
- Less bugs that end in the hands of the consumer

1.1.2 The Goal of Functional Verification

The goal of verification isn't to prove that the design is bug-free as no design can be entirely bug free for all scenarios. What our focus lies on is to make sure the design can accomplish its task successfully according to specifications.

Currently designs' complexity is growing more and more. So, the process of chip fabrication is very time consuming and costs millions. Any defects in the design if caught at an early stage will save time and cost of repeating many processes in the flow. [1]

1.1.3 The Features of SystemVerilog

Back in the 1990's, Verilog was the primary language to perform functional verification of designs but it was small and not very complex at the time. As design complexity increases, better tools are needed which lead to moving to SystemVerilog and research for better and simpler verification methods is continuous.

It's basically Verilog with some added features. Some of these features are:

- Compatible with Object-Oriented Programming
- Generation of constrained-random stimulus
- Functional coverage
- Transaction level modeling

There are many other useful features, but these allow us to create testbenches at a higher level of abstraction than with an HDL or a programming language such as C. [2]

1.2 The Features of UVM

Universal verification methodology (UVM) is a standard to enable faster development and reuse of verification environments and verification IP throughout the industry.

It's a library of basic open-source classes we don't have to know everything about but just include it and extend from it while writing our testbench. For example, think of C coding language if you need to use strings all you have to do is include the <string.h> library without caring about its content.

It is mainly derived from Open Verification Methodology (OVM) and is supported by multiple EDA vendors like Synopsys, Cadence, Mentor, etc.

Some of the built-in UVM classes is shown in (Figure 1.2).

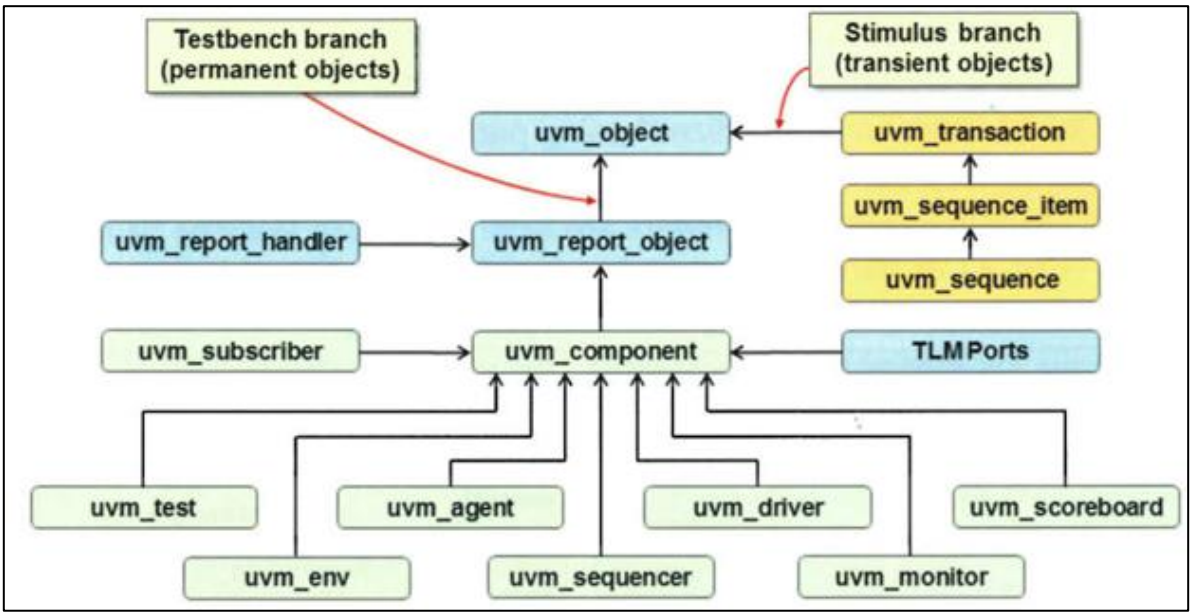


Figure 1.2: main uvm classes

The main idea behind UVM is to help companies develop modular, reusable and scalable testbench structures.

Its class library provides generic utilities like configuration database, TLM and component hierarchy in addition to data automation features like copy, print and compare.

It brings in a layer of abstraction where every component in the environment has one specific role and its code can be written separately.

Hence, all testbench environments has the same structure and components for every IP and can be reused in different projects. [3]

1.3 Goals of the project

Our goal at the end of this project is to build a UVM verification environment for OpenHMC controller RTL design.

Project Objectives are:

1. Learn Verilog and SystemVerilog programming languages
2. Learn OOP
3. Understand basic and advanced verification skills and flow
4. Understanding SystemVerilog Assertions (SVA)
5. Developing codes of a UVM environment
6. Learn how to read a specification file
7. Learn how to read user manual of a simulation tool

1.4 Organization of the book

Chapter 1: “Introduction”, defining functional verification, uvm and the project’s objectives.

Chapter 2: “Introduction About RTL”, highlight important information about the RTL design.

Chapter 3: “HMC Architecture & Specs”, highlight important information extracted from the specs.

Chapter 4: “Verification Plan & Features”, present the developed verification plan.

Chapter 5: “Testbench Architecture”, present the architecture and components of the testbench environment.

Chapter 6: “RF Agent”, present and explain everything about the register file agent in our environment.

Chapter 7: “AXI Agent”, present and explain everything about the AXI agent in our environment.

Chapter 8: “HMC Agent”, present and explain everything related to the HMC agent in our environment.

Chapter 9: “Scoreboard & Coverage Collector”, defining scoreboard and coverage components in UVM and their role in the environment while explaining the ones implemented in our environment.

Chapter 10: “Codes Summary”, summary of the ordering of our codes in github while presenting some statistics.

Chapter 11: “Simulation and Results”, in details information about all the steps taken during simulation and presenting the evolution of the results throughout the tests.

Chapter 12: “Conclusion”, ending the book on all learned points and problems faced during the project.

Chapter 2
Introduction About RTL

2.1 What's OpenHMC?

OpenHMC is an open-source project developed by the Computer Architecture Group (CAG) at the University of Heidelberg in Germany [4]. The OpenHMC controller is presented as a high-level block diagram as in (Figure 2.1). The transmit and receive FIFOs form an AXI-4 streaming interface and allow the user to access the controller from a different clock domain. On the other side, the transceivers transmit and receive packets to/from other transceivers of another OpenHMC controller where a registered output holds the data reordered on a lane-by-lane basis; allowing seamless integration with any transceiver types. A register-file provides access to control and monitor the operation of the controller.

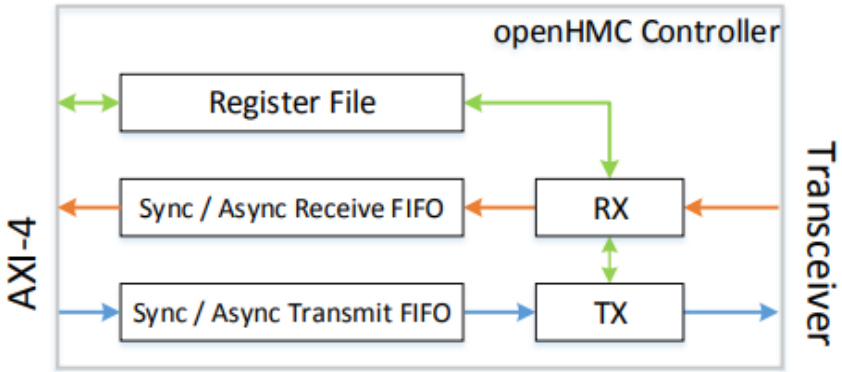


Figure 2.1: OpenHMC controller block diagram

2.2 Features

The OpenHMC host controller implements the following features as described in the HMC specification (HMC Specifications v1.0) [5]:

- Full link-training, sleep mode and link retraining.
- 16Byte up to 128Byte read and write (posted and non-posted) transactions.
- Posted and non-posted bit-write and atomic requests.
- Mode read and write.
- Error response.
- Full packet flow control.
- Packet integrity checks (sequence number, packet length, CRC).

- Full automatic error handling and link retry.

2.2.1 Supported Configurations

Currently the following configurations for the AXI-4 interface are supported:

- 2 FLITs per Word / 256-bit Datapath.
- 4 FLITs per Word / 512-bit Datapath.
- 6 FLITs per Word / 768-bit Datapath.
- 8 FLITs per Word / 1024-bit Datapath.

Any configuration can be applied to any HMC link width and speed.

2.3 Module Description

2.3.1 Top Module (openhmc_top.v)

The OpenHMC top module instantiates and connects all logical sub-modules and does not contain any logic itself. It provides the AXI-4, Transceiver and Register File interfaces. The OpenHMC controller top level includes the two possible clock domains and main interface signals. The host controller is also referred to as 'Requester' and the data flow from host to HMC is called downstream traffic, or transmit direction (TX). The requester issues request packets and receives responses. On the other hand, the HMC is the 'Responder' and any traffic flowing in host direction is called upstream traffic, or receive direction (RX). The responder receives and processes requests, and returns responses if desired by the request type. In the following, all sub-modules are described in the order they are logically passed by a request/response transaction.

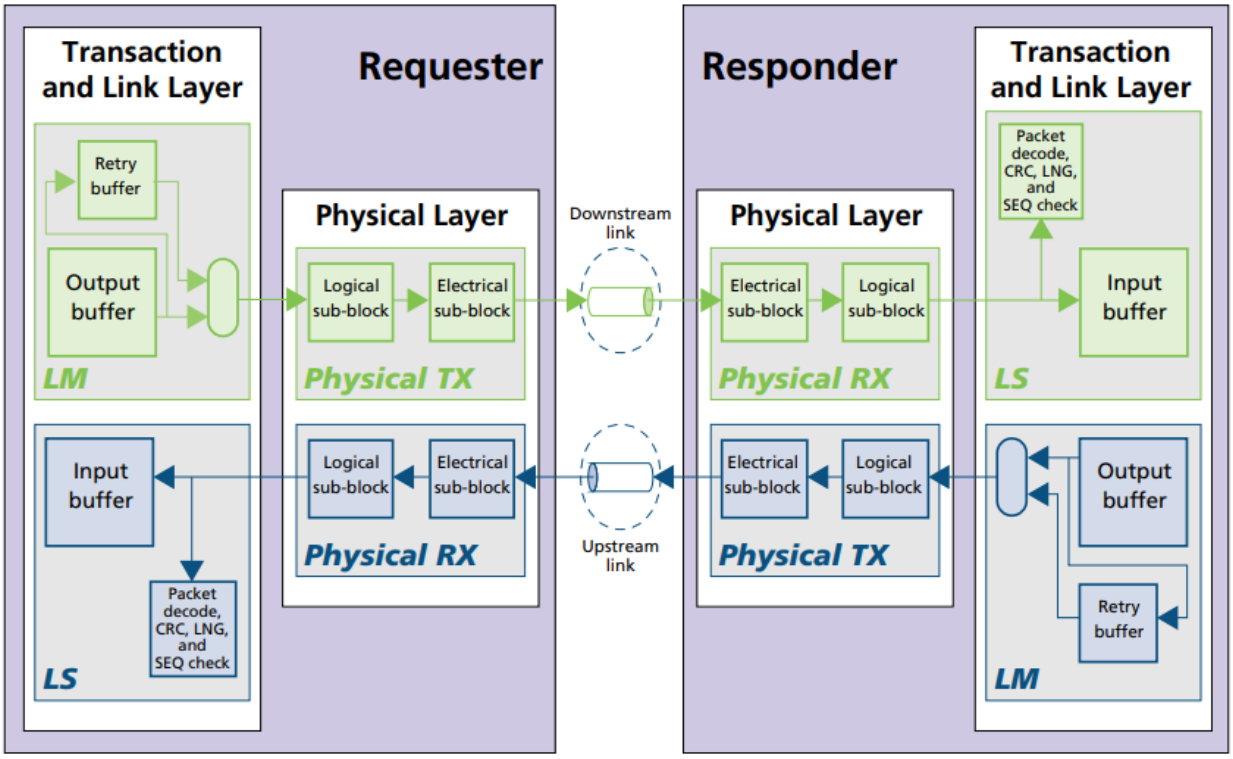


Figure 2.2: Link Data Transmission Implementation Example

2.3.2 Transmit and Receive FIFOs

The transmit and receive FIFOs connect the user logic in the clk_user clock domain to the OpenHMC controller in the clk_hmc clock domain. Both FIFOs appear as an AXI-4 Stream Protocol Interface to the user. The FIFOs can be either configured as synchronous or asynchronous.

2.3.3 TX Link (tx_link.v)

The TX Link has two main interfaces, that is the input FIFO interface to receive HMC packets and the output register stage which provides scrambled and lane-by-lane re-ordered data FLITs to connect the transceivers. The user must generate HMC packets within the user logic, including the 64bit header. Also, the user is responsible for operational closure using TAGs, if desired. Note that an unsupported command or a dln/lng mismatch may produce undefined behavior in the current implementation. The 64bit tail must be set all to zero since it will be filled in the TX Link. Internally, the openHMC controller uses register stages to encapsulate logically-independent units, and to avoid critical paths due to excessive use of combinational logic. The main control function is implemented as a Finite State Machine (FSM) as shown in (Figure 2.3).

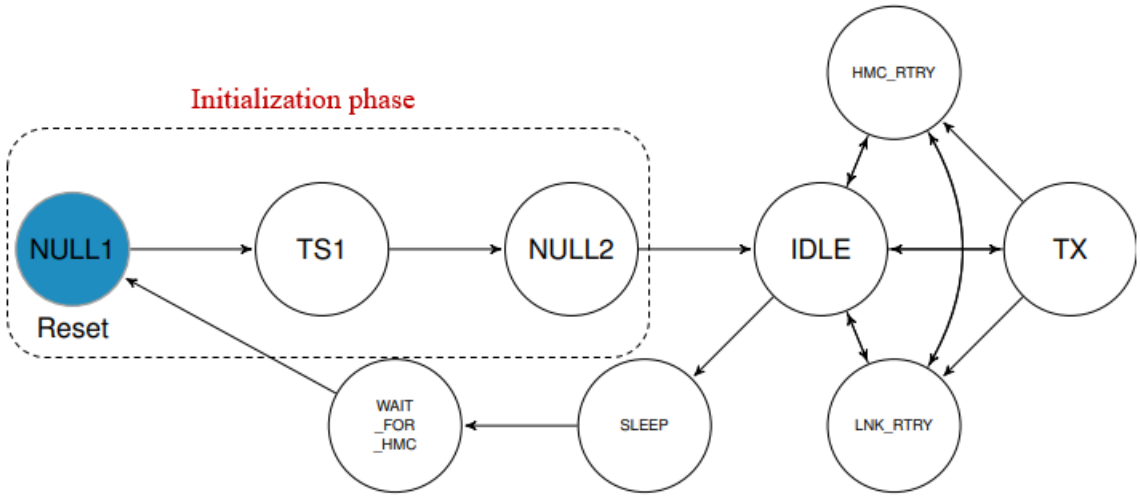


Figure 2.3: The Design Finite State Machine (FSM)

States and transitions are listed in (Table 2.1) and (Table 2.2). The next states are listed in the order of their priority. By default, the current state is maintained. When in TX state FLITs are processed as implied by the blue path in (Figure 2.4). Register File (RF) signals and such that are driven by the RX link are represented by green colored, control signals by gray colored lines. The operation of the TX link can be summarized as follows: First, data FLITs are collected at the FIFO interface. A token handler keeps track of the remaining tokens in the HMC input buffer. With each FLIT transmitted the token count is decremented. When the token count is sufficient and no other interrupt occurs the Return Token Count (RTC) is added to return tokens to the HMC, which indicates the number of FLITs that passed the RX input buffer. Afterwards, the Sequence Number (SEQ) and the Forward Retry Pointer (FRP), which is also the retry buffer read pointer are added. At this point all FLITs are also written to the retry buffer. If there is a link retry request (signaled by tx_link_retry_request) data is retransmitted out of the retry buffer. instead of the regular datapath. Eventually the Return Retry Pointer (RRP) which is the last received HMC FRP is added and the CRC generated. Data is then scrambled and reordered on a lane-by-lane basis depending on the configuration (NUM_LANES and FPW). (Figure 2.5) shows an example for a 512-bit / 8-lane configuration where each transceiver connects to 64bit of the parallel output stage.

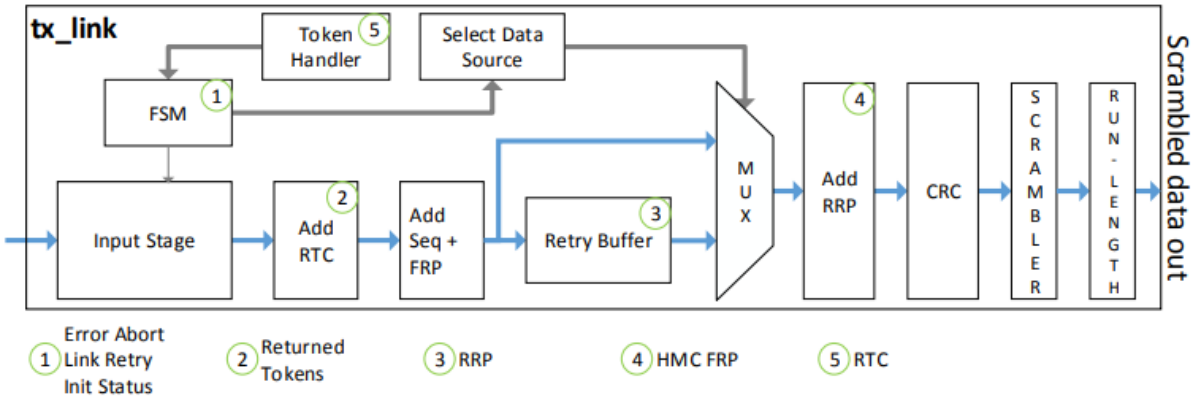


Figure 2.4: TX link diagram

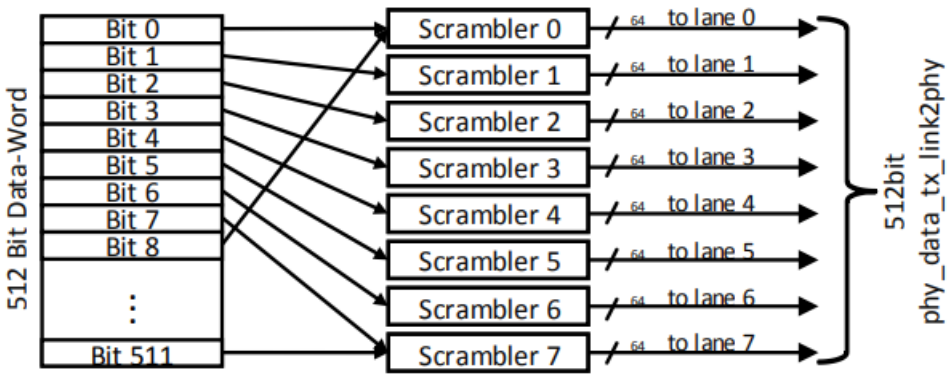


Figure 2.5: Data-Reordering: 4FLIT/512bit example

Table 2.1: TX FSM State Table

State	Description
NULL1	Transmit NULL FLITs (Reset State)
TS1	Transmit the lane dependent TS1 sequence
NULL2	Transmit NULL FLITs
IDLE	Send TRET packet if there are tokens to be returned
TX	Transmit packets
HMC_RTRY	Send start retry packets
LNK_RTRY	Send clear retry packets and perform link retry
SLEEP	Set LXRXPS = low to request HMC sleep mode
WAIT_FOR_HMC	Wait until corresponding LXTXPS pin is high to exit sleep mode

Table 2.2: TX FSM Transition Table

State	Next State & Trigger
NULL1	TS1: RX received NULL FLITs
TS1	NULL2: RX descramblers aligned
NULL2	IDLE: link_is_up
IDLE	HMC_RTRY: force_hmc_retry LNK_RTRY: tx_link_retry_request SLEEP: rf_hmc_sleep TX: retry_buffer !full and tokens are available
TX	HMC_RTRY: force_hmc_retry LNK_RTRY: tx_link_retry_request IDLE: no more data to transmit
HMC_RTRY	LNK_RTRY: tx_link_retry_request TX: retry_buffer !full and tokens are available IDLE: no more data to transmit
LNK_RTRY	HMC_RTRY: force_hmc_retry TX: retry_buffer !full and tokens are available IDLE: no more data to transmit
SLEEP	WAIT_FOR_HMC: as rf_hmc_sleep_requested is de-asserted
WAIT_FOR_HMC	NULL1: as hmc_LXTXPS transitions to high

2.3.3.1 TX Retry Buffer (openhmc_ram.v)

The retry buffer holds a copy of each FLIT transmitted for possible retransmission. NULL FLITs and flow packets, except TRET, are not subject to flow control and retransmission and are not stored to the retry buffer. The retry buffer actually consists of FPW times 128-bit RAMs so that each FLIT can be addressed independently. One address (i.e., the FRP) is generated for each packet header. Since the required and accumulated RAM space is defined by the pointer size (FRP = RRP = 8 bit = 256 FLITs), the depth per RAM in this implementation is defined as 256 entries divided by FLITs Per Word (FPW). (Table 2.3) summarizes the RAM properties for different data-width configurations. Note that a 6-FLIT configuration results in reduced RAM capacity since 6 is not a power of 2 and therefore the next higher of LOG_FPW must be chosen leaving some addresses

unused. The least significant bits address the target RAM while the remaining bits refer to a specific FLIT within that RAM. The entire value is called FRP, and at the same time is the RAM write pointer. As a result of this addressing scheme, FRPs are not generated consecutively but still incremental, as packets may consist of more than one FLIT. The read pointer of the RAM moves with each RRPs received at the RX Link, following the write pointer and therefore excluding potential FLITs from retransmission.

Table 2.3: RAM Configurations

Datawidth in FPW	Depth per RAM [bits/entries]
2	7/128
4	6/64
6	5/32
8	5/32

2.3.3.2 Scrambler (tx_scrambler.v)

Scramblers use a Linear Feedback Shift Register (LFSR) to ensure Clock-Data Recovery (CDR) over high-speed serial links and replace encodings such as 8b/10b. One scrambler per lane is initialized and its LFSR preloaded with a lane-specific seed.

2.3.3.3 Lane Run Length Limiter (tx_run_length_limiter.v)

The HMC specification defines a maximum of 85 bits per lane without a logical transition to ensure CDR. When a lane reaches this limitation, a transition must be forced to so that the receiver's Phase-Locked Loops (PLLs) stay locked. The granularity of the run length limiter is adjustable and can be set depending on die area and speed requirements (generally: lower granularity = more logic and area utilization). Also consider technological conditions when determining the best value, e.g. which Look-Up Tables (LUTs) are used.

2.3.3.4 CRC (tx_crc_combine.v)

The CRC architecture was specifically chosen to scale with different data-widths. It consists of one 128-bit CRC per FLIT (crc_128_init). While the CRCs are calculated another part of the logic assigns the targeted CRC to the tail of the corresponding packet. After the CRCs are calculated all 32-bit remainder that belong to the same packet are shifted to a dedicated accumulation CRC stage (crc_accu). These remainders form the actual CRC within a single cycle. Finally, the output CRCs are added to the tail of the packets.

2.3.3.5 General Notes on TX Link

The TX link only returns one flow packet per cycle which is sufficient and an easy way to save some logic. However, (re-)initialization for instance will take some additional cycles to transmit

all available tokens since only 31 tokens may be returned within a single Token Return (TRET) packet.

2.3.4 RX Link (rx_link.v)

The RX Link receives responses issued by the HMC. It then performs data integrity checks, unpacks all valid and required information out of header and tail and forwards the information to the TX Link. Only valid FLITs that pass all checks will enter the input buffer and can be collected at the AXI-4 slave interface. (Figure 2.6) shows a block diagram of the RX Link where the data flow is indicated by orange, signals to the TX Link and to the RF by green, and control signals by gray colored arrows. Note that the regular datapath is only selected after link initialization is done.

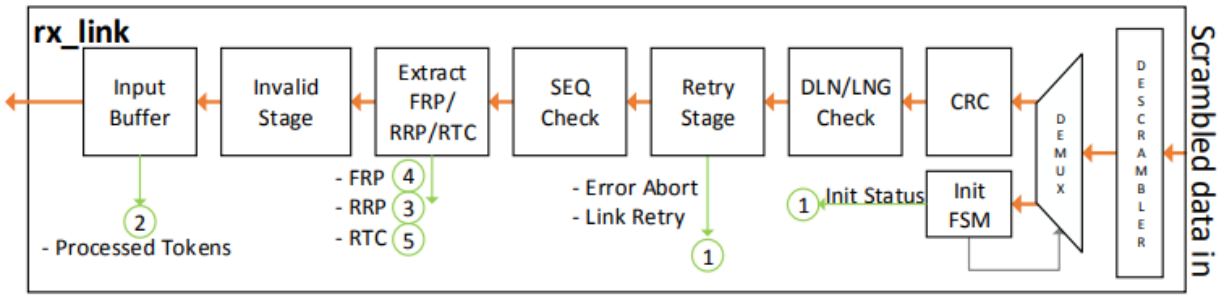


Figure 2.6: RX link diagram

For this purpose, the initialization FSM controls a De-Multiplexer (DEMUX) to distribute input data. Initialization in the RX link is divided into separate stages. As soon as the phy_rx_ready signal at the openhmc_top module is set, initialization begins. The RX link first waits for alle descramblers to lock. Afterwards lane polarity is detected (if parameter DETECT_LANE_POLARITY=1) and applied (if CTRL_LANE_POLARITY=1). As the HMC begins to send TS1 sequences the RX link first bit_slips each lane individually until a valid TS1 is seen.

Then it looks for the lane that is most/least advanced (parameter: BIT_SLIP_SHIFT_RIGHT) and aligns all other lanes accordingly.

2.3.4.1 Descrambler (rx_descrambler.v)

The rx_descrambler module is instantiated once per lane and is self-seeding, which means that it automatically determines the correct value for the internal LFSR. As the seed for a descrambler is determined the descrambler is considered locked. Additionally, each descrambler expects a dedicated, so called 'bit_slip' single bit input which is used compensate lane to lane skew. When bit_slip is set, input data on the specific lane is delayed by one bit during initialization. This procedure is applied until all descramblers are fully aligned / synchronous to each other.

2.3.4.2 CRC (rx_crc_compare.v)

The rx_crc_compare module is very similar to the tx_crc_combine instantiated in the TX Link. The biggest difference is that the CRCs are not added to the tail of a packet at the end of the data pipeline but compared instead. The corresponding poisoned or error flag for the tail of the faulty

packet is set if a mismatch occurs. Additionally, the data pipeline of this module holds information bits for valid/header/tail FLITs as this information will be used in the RX link.

2.3.4.3 Input Buffer

The input buffer holds 2**LOG_MAX_RX_TOKENS entries, where each entry is as wide as the datapath (DWIDTH). This results in more resource utilization, but allows a series of 2**LOG_MAX_RX_TOKENS cycles, carrying one valid FLIT each to be shifted-in without a need for additional buffer distribution and utilization logic. Each valid FLIT at the buffer output returns 1 token to the TX link on a shift_out event. These tokens will be returned as RTC to the HMC. OpenHMC does not forward poisoned packets to the input buffer.

2.3.5 Register File (openhmc_rf.v)

The Register File features three main types of registers: Control, Status, and Counter. Control registers directly affect OpenHMC or HMC operation. Status registers can be used to monitor the OpenHMC status, especially during initialization. Counters allow performance measurement. For a full list of available registers, see Appendix B. Note that there are several 'reserved' fields which are not listed in the table of registers. These reserved fields provide some space to add additional information and also serves as a byte aligner for other fields. Reserved bits will be tied to constant 0 during synthesis.

2.3.6 Header File

The following header files are present: hmc_field_functions.h hmc_field_functions contain useful functions that return fields such as the packet length or the CRC out of HMC headers or tail.

Chapter 3
HMC Architecture & Specs

3.1 Interfaces, pins and configuration parameters

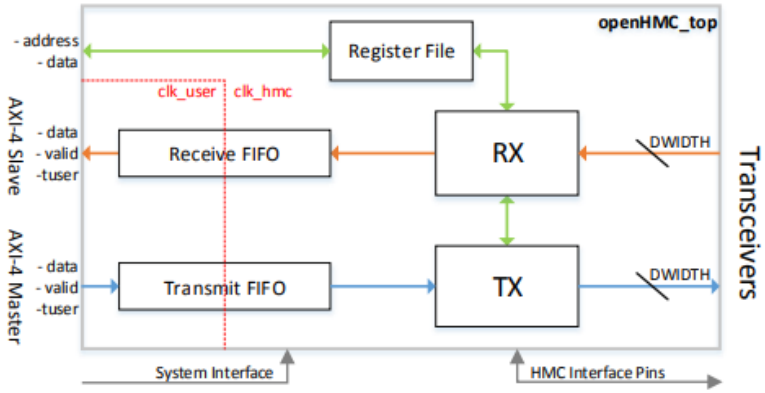


Figure 3.1: OpenHMC controller architecture in details

3.1.1 Configuration parameters

We have some configuration parameters in the top module which are shown in (Table 3.1). [4]

Table 3.1: Configuration parameters in top module

Parameter	Default value	Meaning of parameter
FLIT_SIZE	128	Size of Flow units in bits
FPW	4	Number of FLITS* per word
DWIDTH	FPW*128	Word length
NUM_LANES	8	Number of lanes in the transceiver side
LOG_NUM_LANES	3	Log_2(NUM_LANES)
NUM_DATA_BYTES	FPW*16	Defines the AXI-4 TUSER bus width in bytes.
HMC_RF_WWIDTH	64	Width of word to be written in register file
HMC_RF_RWIDTH	64	Width of word to be read from register file
HMC_RF_AWIDTH	4	Width of addresses of registers in the register file
LOG_MAX_RX_TOKENS	8	Set the depth of the RX input buffer
LOG_MAX_HMC_TOKENS	10	Set the depth of the HMC input buffer
HMC_RX_AC_COUPLED	0	Set to 0 to bypass the run length limiter
DETECT_LANE_POLARITY	0	Set to 0 if lane polarity is not applicable
CTRL_LANE_POLARITY	0	Set to 0 if lane polarity is not applicable
CTRL_LANE_REVERSAL	0	Set to 0 if lane reversal is not applicable
CTRL_SCRAMBLERS	1	Set to 0 to enable (de-)scramblers for debugging
OPEN_RSP_MODE	0	Set to 1 if running response open loop mode, bypasses the RX input buffer
RX_RELAX_INIT_TIMING	1	Save resources and erases timing closure. Lane reversal is still detected
RX_BIT_SLIP_CNT_LOG	5	Define the number of cycles between bit slips
SYNC_AXI4_IF	1	Set to 1 if AXI IF is synchronous to clk_hmc
XIL_CNT_PIPELINED	0	If Xilinx counters are used, set to 1 to enabled output register pipelining
BITSLIP_SHIFT_RIGHT	1	Define how the parallel data is shifted by bit slip.
DBG_RX_TOKEN_MON	1	Set to 0 to remove the RX Link token monitor
CLK_PERIOD	4.27	The clock cycle period

3.1.2 Interfaces and Pins

We have 5 interfaces in our design.

3.1.2.1 AXI Interface

The AXI (Advanced eXtensible Interface) bus is an on-chip communication bus protocol developed by ARM. AXI is a high-performance, point-to-point, master-slave parallel bus used to connect on-chip peripheral circuits (or IP blocks) to processor cores. AMBA AXI4 (Advanced eXtensible Interface 4) is the fourth generation of the AMBA interface specification from ARM. The AXI-4 protocol is an update to AXI-3 which is designed to enhance the performance and utilization of the interconnect when used by multiple masters. [4]

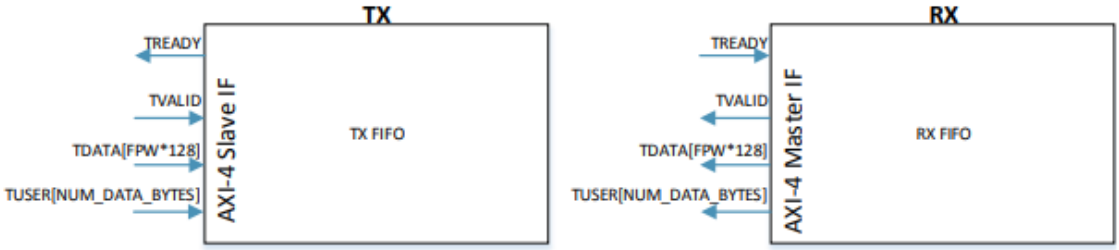


Figure 3.2: AXI-4 interface diagram

The openHMC controller provides AXI-4 stream protocol interfaces for TX and RX. Figure 3.2 provides the AXI-4 TX and RX interfaces diagram used in this design. The use and the corresponding size of the signals in this interface is described below. [4]

TREADY [1 bit]
• TX: openHMC is ready to sample TDATA and TUSER.
• RX: User application is ready to sample TDATA and TUSER.

TVALID [1 bit]
• TX: TDATA and TUSER are sampled on TX when TVALID=1 and TREADY=1. TVALID may be held high even when TREADY=0.
• RX: TDATA and TUSER are valid when TVALID=1. TREADY may be held high even when TVALID=0. TDATA and TUSER will not change when TREADY=0.

TDATA [FPW*FLIT_SIZE]
The TDATA bus expects complete HMC request packets, starting with the 64bit header followed by data FLITs. Note that a single AXI cycle can carry (parts of) multiple packets on both interfaces, TX and RX. The user is responsible to populate all request header fields (will be explained in the HMC request packets fields & commands). Note that the TAG field is optional, but required for operational request/response closure. The tail must be set to all zeroes (The openHMC controller expects complete HMC packets at the TX interface, and outputs such at the RX interface. HMC request packets must be generated within the user logic, i.e. set a command, lng/dln fields, the cube ID and target address. The tail must be set to zero). Packets may start at any 128-bit/ FLIT border. 'Bubbles' between packets are allowed as long as the corresponding valid bit(s) is/are kept low. All FLITs of a packet must be transmitted throughout consecutive FLITs. Also, when a packet spreads over multiple 512-bit cycles, TVALID must be held high until the entire packet (including

its tail) is transmitted. On RX, the openHMC controller outputs complete HMC response packets. Data is valid when TVALID=1 and the output will not change while TREADY=0. Contrary to TX, the user has full control on the assertion of TREADY. When a response header appears, the packet does not need to be sampled consecutively throughout its tail.

TUSER [NUM_DATA_BYTES bit]
The user is responsible to set the information on the TX TUSER bus respectively, the controller provides this information at the RX TUSER bus. Note that only a part of the TUSER bus is populated. That is 3*FPW bits on TX and 4*FPW on RX.
valid at TUSER index [FPW-1:0]: Valid FLIT indicator (including header and tail), one bit per FLIT.
hdr at TUSER index [(2*FPW)-1:FPW]: Header indicator, one bit per FLIT.
tail at TUSER index [(3*FPW)-1:2*FPW]: Tail indicator, one bit per FLIT
err_rsp [only on RX] at TUSER index [(4*FPW)-1:3*FPW]: Indicates an error response packet at the corresponding FLIT position. One bit per FLIT. Error response packets are single FLIT packets and have all flags (valid/hdr/tail/err_rsp) set Every FLIT on the TDATA bus corresponds to one bit in the valid, hdr, and tail fields on TUSER. FLIT 0 at TDATA[127:0] is defined by valid[0] (TUSER[0]), hdr[0](TUSER[FPW]), and tail[0](TUSER[2*FPW]).
Example: TDATA holds a header on FLIT position 0 (TDATA[127:0]). Set hdr[0] respectively TUSER[FPW] to 1. Since a header is a valid FLIT, set valid[0] / TUSER[0] to 1. This scheme applies to all FLITs on the TDATA bus. Figure 3.4 illustrates how to set the TUSER signal according to the content of the TDATA bus in Figure 3.3. [4]

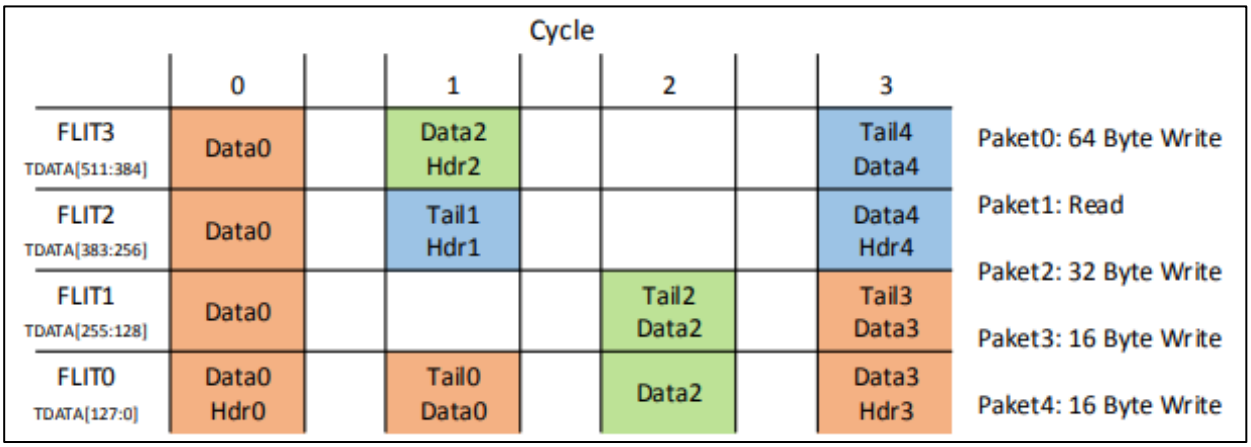


Figure 3.3: Example transactions on the AXI TX TDATA bus for FPW=4

	Cycle						
	0		1		2		3
Tail TUSER[11:8]	4'b0000		4'b0101		4'b0010		4'b1010
Hdr TUSER[7:4]	4'b0001		4'b1100		4'b0000		4'b0101
Valid TUSER[3:0]	4'b1111		4'b1101		4'b0011		4'b1111
TUSER[11:0]	0x01F		0x5CD		0x203		0xA5F

Figure 3.4: TUSER Example for FPW=4

3.1.2.2 HMC Physical Layer (transceiver) interface

The TX Link provides a **DWIDTH** wide register output **phy_data_tx_link2phy** with scrambled and lane-by-lane ordered data, driven by **clk_hmc**. Hence the bits [(1*LANE_WIDTH)-1:(0*LANE_WIDTH)] contain data for lane 0, [(2*LANE_WIDTH)-1:(1*LANE_WIDTH)] data for lane 1 and so on. An additional input **phy_ready** should be connected to transceivers **'reset_done'** (or similar) to allow monitoring of the transceiver status. The RX Link's data input register **phy_data_rx_phy2link** expects input data by the receivers using the same ordering as explained for the TX Link. Lane reversal is detected and applied in the RX Link and does not affect ordering. Additionally, the RX Link outputs **bit_slip** wires, one per lane, used to compensate lane-to-lane skew on the parallel input data during initialization. Connect these to the corresponding transceiver. If **lane_polarity** is performed within the transceivers, the **phy_lane_polarity** output must be used. **'CTRL_LANE_POLARITY'** must be set to 1 in this case. For **CTRL_LANE_POLARITY**=0, **phy_lane_polarity** is tied to 0. All signals are summarized in Table 3.2. [4]

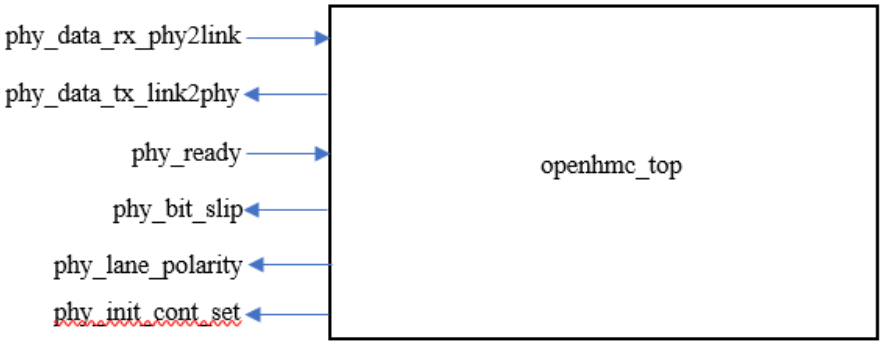


Figure 3.5: HMC physical layer (transceiver) interface diagrams with pins

Table 3.2: HMC physical layer (transceiver) interface signals

Signal	Width	Description
phy_data_tx_link2phy	DWIDTH	Lane by lane ordered output.
phy_data_rx_phy2link	DWIDTH	Lane by lane ordered input.
phy_ready	1	Signalize that the transceivers are ready.
phy_bit_slip	NUM_LANES	Bit_slip is used to compensate lane to lane skew. Bit_slip is controlled by the rx_link for each lane individually.
phy_lane_polarity	HMC_ NUM_ LANES	connect transceiver polarity inputs if polarity is controlled within the transceivers and **'CTRL_LANE_POLARITY'** is set to 1.

3.1.2.3 System Interface

The controller top module (openhmc_top) expects a clock and a reset per clock domain. Most likely, clk_hmc and the parallel transceiver clock domain will be sourced by the same driver. The user clock clk_user may be any frequency equal to or higher the frequency of clk_hmc. Therefore, both clocks can origin from the same source. If SYNC_AXI4_IF is set to 0, source both clocks from the same source. Figure 3.6 shows the system interface. Note that both resets are active low. [4]

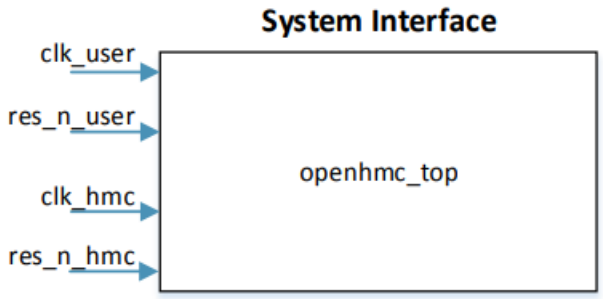


Figure 3.6: System interface diagram with its pins

3.1.2.4 HMC Interface

The HMC provides the four signals presented in (Figure 3.7). All signals are active low. [4]

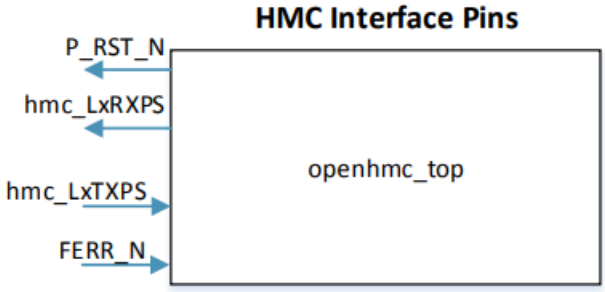


Figure 3.7: HMC interface diagram pins

3.1.2.5 Register File Interface

A Register File module allows to control and monitor the openHMC operation. The interface signals are shown in (Figure 3.8) and described in (Table 3.3). First the target address must be applied. For a write, write_data must hold the 64-bit value to be written.

Data is sampled when write_enable is asserted. For a read the read_enable signal must be asserted instead. Each operation is confirmed by the access_complete signal set for one cycle. In case that an invalid address was applied, invalid_address will remain as long as read_en or write_en are active. The user must not assert write_en and read_en both at the same time. The RF resides in the clk_hmc clock domain and uses the active low res_n hmc reset signal. (Figure 3.9) provides an example for a register write followed by a read to address 0x10. [4]

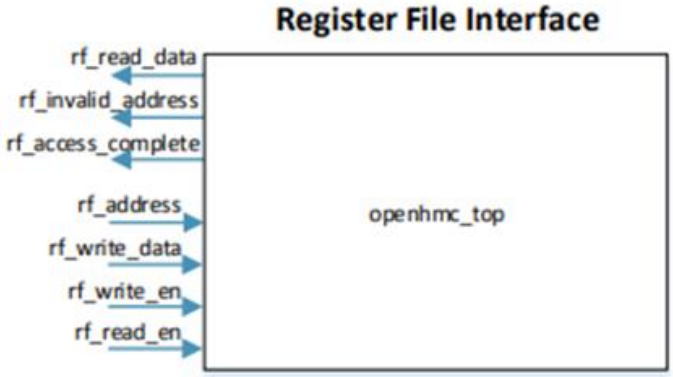


Figure 3.8: Register file interface diagram with its signals

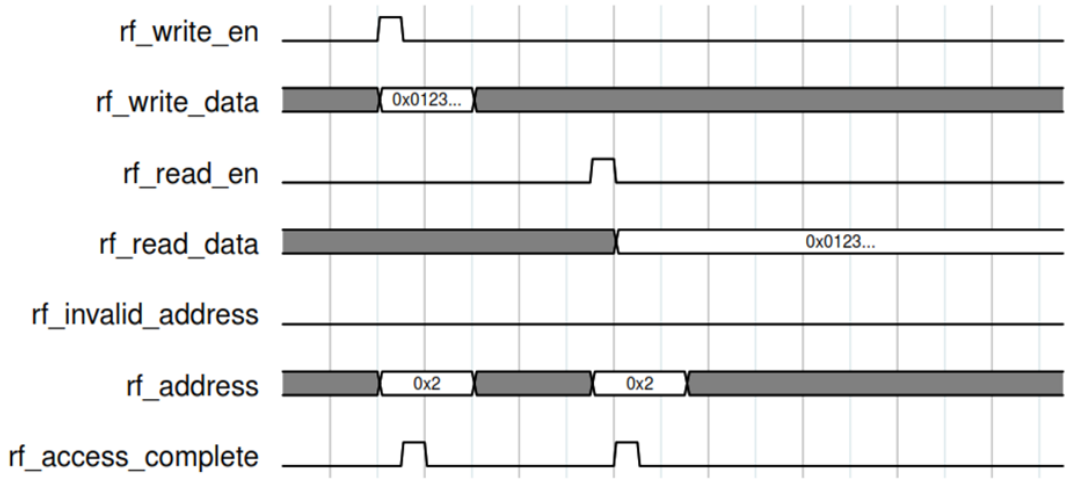


Figure 3.9: Register File Access: Write and read register 0x2

Table 3.3: Register file interface signals

Signal	Width	Description
rf_write_data	HMC_RF_WWIDTH	Value to be written.
rf_read_data	HMC_RF_RWIDTH	Requested Value. Valid when access_complete is asserted.
rf_address	HMC_RF_AWIDTH	Address to be read or written to.
rf_read_en	1	Read the address provided.
rf_write_en	1	Write the value of write_data to the address provided.
rf_invalid_address	1	Address out of the valid range.
rf_access_complete	1	Indicates a successful operation.

3.2 Register File contents and its address map

It contains a total of 10 register. 3 main registers for monitoring status and control, 6 are counters for sent and received packets and for any errors occurred and one to reset all counters. All their contents is shown in the following tables [4]

Table 3.4: Register file address map

Register	Address	Description
status_general	0x0	General HMC Controller Status.
status_init	0x1	Debug register for initialization.
control	0x2	Control register
sent_p	0x3	Number of posted requests issued
sent_np	0x4	Number of non-posted requests issued
sent_r	0x5	Number of read requests issued
poisoned_packets	0x6	Number of poisoned packets received
rcvd_rsp	0x7	Number of responses received
counter_reset	0x8	Reset all counter
tx_link_retries	0x9	Number of Link retries performed on TX
errors_on_rx	0xA	Number of errors seen on RX
run_length_bit_flip	0xB	Number of bit flips performed due to run length limitation
error_abort_not_cleared	0xC	Number of error_abort_mode not cleared

Table 3.5: status_general register

Field	Bit	Width [Bits]	Description & Encoding	Res
link_up	0	1	Link is ready for operation	0
link_training	1	1	Link training in progress	0
sleep_mode	2	1	HMC is in Sleep Mode	0
FERR_N	3	1	HMC FERR_N signal	0
lanes _reversed	4	1	0: Normal Operation 1: Lanes reversed (lane 15/8 with 0, ...)	0
phy_tx_ready	8	1	SerDes TX reset is done	0
phy_rx_ready	8	1	SerDes RX reset is done	0
hmc_tokens _remaining	16+: LOG_MAX_HMC_ TOKENS	LOG_MAX_ HMC_ TOKENS	Number of Tokens remaining in the HMC input buffer	0
rx_tokens _remaining	32+: LOG_MAX_RX_ TOKENS	LOG_MAX_ RX_ TOKENS	Number of Tokens remaining in the rx_link input buffer (if param DBG_RX_TOKEN_MON = 1)	0
lane _polarity _reversed	48+: NUM_LANES	NUM LANES	0: Normal Operation 1: Data is logically inverted lane-by-lane	0

Table 3.6: status_init register

Field	Bit	Width [Bits]	Description & Encoding	Res
lane _descramblers _locked	0+: NUM_LANES	NUM_LANES	Lane by lane descrambler locked	0
descrambler _part _aligned	16+: NUM_LANES	NUM_LANES	Lane by lane descrambler partially aligned	0
descrambler _aligned	32+: NUM_LANES	NUM_LANES	Lane by lane descrambler fully aligned	0
all_descramblers _aligned	48	1	All descramblers are aligned	0
status_init_rx _init_state	51:49	3	Init status of the RX link 3'b000: HMC_DOWN 3'b001: HMC_WAIT_FOR_NULL 3'b010: HMC_NULL 3'b011: HMC_TS1_PART_ALIGN 3'b100: HMC_TS1_FIND_REF 3'b101: HMC_TS1_ALIGN 3'b110: HMC_NULL_NEXT 3'b111: HMC_UP	0
status_init_tx _init_state	53:52	2	Init status of the TX link 2'b00: INIT_TX_NULL_1 2'b01: INIT_TX_TS1 2'b10: INIT_TX_NULL_2 2'b11: INIT_DONE	0

Table 3.7: Other Counters (Each Entry equals one Register)

Field	# Bits	Description & Encoding	Reset
tx_link_retries	32	Incremental 1-bit counter: Number of Link retries performed on TX	0
errors_on_rx	32	Incremental 1-bit counter: Number of successful HMC retries performed	0
run_length _bit_flip	32	Incremental 1-bit counter: How many bit_flips were performed by the run length limiter	0
error_abort _not_cleared	32	Incremental 1-bit counter: Indicates the number of link retry attemps that timed out	0
counter_reset	1	Reset counter in the 'Other Counter' category. This bit is automatically cleared	0

Table 3.8: Performance Counters (Each Entry equals one Register)

Field	# Bits	Description & Encoding	Reset
poisoned_packets	64	Number of poisoned packets received	0
sent_np	64	Number of non posted requests issued (including all types)	0
sent_p	64	Number of Posted Data Write requests issued	0
sent_r	64	Number of Read Data requests issued	0
rcvd_rsp	64	Number of responses received	0

Table 3.9: Control register

Field	Bit	Width[Bits]	Description & Encoding	Res
p_rst_n	0	1	Active low HMC reset	0
hmc_init_cont_set	1	1	Allow descramblers to lock	0
set_hmc_sleep	2	1	Request HMC sleep mode. Sleep mode can be monitored by the 'sleep_mode' field in the status_general register	0
scrambler _disable	3	1	Disable Scrambler and Descrambler for testing purposes	0
run_length _enable	4	1	Disable the run length limiter in the TX scrambler logic	0
first _cube_ID	7:5	3	Set the Cube ID of the first HMC connected. Used in irtry packets	0
debug _dont_send_tret	8	1	Prohibit controller from sending any TRET packets	0
debug _halt_on _error_abort	9	1	HALT tx_link after rx_link entered error abort	0
debug _halt_on _tx_retry	10	1	HALT tx_link after performing a ret	0
rx_token _count	16+: LOG_MAX_RX_ TOKENS	10	Set the input buffer space in the RX block	{LOG_MAX_ RX_TOKENS {1'b1}}
irtry_received _threshold	36:32	5	Set the number of irtry packets to be received to trigger retry	0x10
irtry_to _send	44:40	5	Set the number of irtry to be sent	0x18

3.3 TS1 (Training Sequence) Definition

The definition and notes about the table in shown in (Table 3.10) and how it is ordered through the lanes in (Table 3.11). [5]

Table 3.10: TS1 bit position

				TS1 bit position											
(Sent last) 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	(Sent first) 0
1	1	1	1	0	0	0	0	L	L	L	L	S	S	S	S
0xF				0x0				Lane dependent [1]				Sequence [2]			

Notes:
1. Lane field = 0xC for lane 7, 0x3 for lane 0, and 0x5 for all other lanes.
2. Sequence[3:0] is a four bit counter that increments from 0 to 15 with each successive TS1 sent on a lane. Sequence value will roll back to 0 after 15 is reached.

Table 3.11: TS1 (Training Sequence) Example

TS1 Ordering	Lane 0	Lane 1-6	Lane 7
0	16'hF030	16'hF050	16'hF0C0
1	16'hF031	16'hF051	16'hF0C1
2	16'hF032	16'hF052	16'hF0C2
3	16'hF033	16'hF053	16'hF0C3
…	…	…	…
14	16'hF03E	16'hF05E	16'hF0CE
15	16'hF03F	16'hF05F	16'hF0CF

3.4 Lane Run Length Limitation

For each HMC RX lane, the scrambled data pattern must meet a run length limit of 85 UI. This is the maximum number of consecutive identical digits allowed. A run of 86 or more consecutive ones (or zeroes) must not be generated by the transmitter on any of the downstream lanes at any time. This restriction is in place so that the clock recovery circuit at each HMC RX lane meets its minimum required transition density to assure correct data alignment. This restriction does not require a particular implementation of the transmitter logic, as long as the run length limit is met at the HMC RX lane inputs. Although a given host may not have a run length limit on the scrambled upstream data, the HMC is designed to meet run length limitations of 85 UI on each lane at all times. This is implemented through the HMC's ability to monitor the scrambled data at each TX

lane. If the scrambled TX data on any lane exceeds 85 consecutive digits, the TX scramble logic forces at least one transition. Forcing a transition corrupts the upstream response and results in a link retry. The probability of such an event is approximately 2-80 for random payload data. The HMC run length limitation feature may be disabled within the mode register. [5]

3.5 Lane Reversal

In order to accommodate different external link routing topologies, the RX logical sub-block has the capability to reverse the lane bit number assignment. Lane reversal at the HMC is detected during initialization when receiving the TS1 training sequence and is automatically compensated for by logic in each receiving logical sub-block that reassigns external Lane 0 to connect to Lane 7 internally, Lane 1 to connect to Lane 6 internally, and so on. Once lane ordering is detected and set by the HMC logical sub-block, it remains unchanged thereafter. The lane reversal mode does not have to be the same for both directions of the link. An HMC component may have one or more links with lane reversal enabled at their respective receivers. The host is responsible for any implementation of lane reversal from the upstream lanes. [5]

3.6 Lane Polarity

In order to accommodate different external Link routing topologies, the RX logical sub-block has the capability to logically invert the received data on a lane-by-lane basis. The polarity is determined during link initialization and remains unchanged thereafter. The training sequence is used to infer polarity at the HMC receiver. The requester is responsible for any implementation of lane polarity inversion from upstream lanes. [5]

3.7 Link Layer

The link layer handles communication across the link not associated with the transaction layer. This consists of the transmission of NULL FLITs and flow packets. All FLITs of a packet must be transmitted sequentially, without interruption, across the link. NULL FLITs are generated at the link layer when no other packets are being transmitted. A NULL FLIT is an all-zeros pattern that is scrambled prior to transmission. Any number of packets can be streamed back-to-back across the link, or they can be separated by NULL FLITs, depending upon system traffic and transaction layer flow control. There is no minimum or maximum requirement associated with the number of NULL FLITs transmitted between packets. NULL FLITs are not subject to flow control. The first nonzero FLIT following a NULL FLIT is considered to be the first FLIT of a packet. Note that data FLITs within a packet may be all zeros, but these lie between a header FLIT and a tail FLIT. Flow packets are generated by the link master to pass flow control and retry control commands to the opposite side of the link. Flow packets are sent when no other link traffic is occurring or when a link retry sequence is to be initiated. Flow packets are single-FLIT packets with no data payload and are not subject to flow control. Flow packets utilize the same header and tail format as request packets, but are not considered requests, and do not have corresponding response commands. [5]

3.8 Transaction Layer

All information transmitted across the links is included in packets that consist of one or more FLITs. A packet always includes an 8-byte header at the beginning of the packet and an 8-byte tail at the end of the packet. The header includes the command field that identifies the packet type, other control fields, and when required, the addressing information. The tail includes flow and link-retry control fields along with the CRC field. There are three different packet types. The first two, request and response packets, are used at the transaction layer. The third type, the flow packet which is only used at the link layer. [5]

1. Request packets are issued by the requester (host or HMC configured as a pass-thru link). The request packet header includes address information to perform a READ or WRITE operation. Write request packets include data FLITs.
2. Response packets are issued by the responder (HMC configured as a host link). Response headers do not include address information. READ responses include data FLITs, and can optionally include write response tags embedded in the header. Write response packets do not include data FLITs.
3. Flow packets are not part of the transaction layer protocol and are not subject to flow control. They are only used at the link to pass flow control and retry control information back to the link master when normal transaction layer packets are not flowing in the return direction. Flow packets are generated at the link master and are decoded and then dropped at the link slave, they are not forwarded and do not pass through the HMC. Each packet type has its own defined header and tail formats, described below. Packets that consist of multiple FLITs are transmitted across the link with the least-significant FLIT number first. The first FLIT includes the header and the least significant bits (LSBs) of the data. Subsequent data FLITs progress with more significant data following. Figure 3.10 illustrates the layout for packets with and without data.

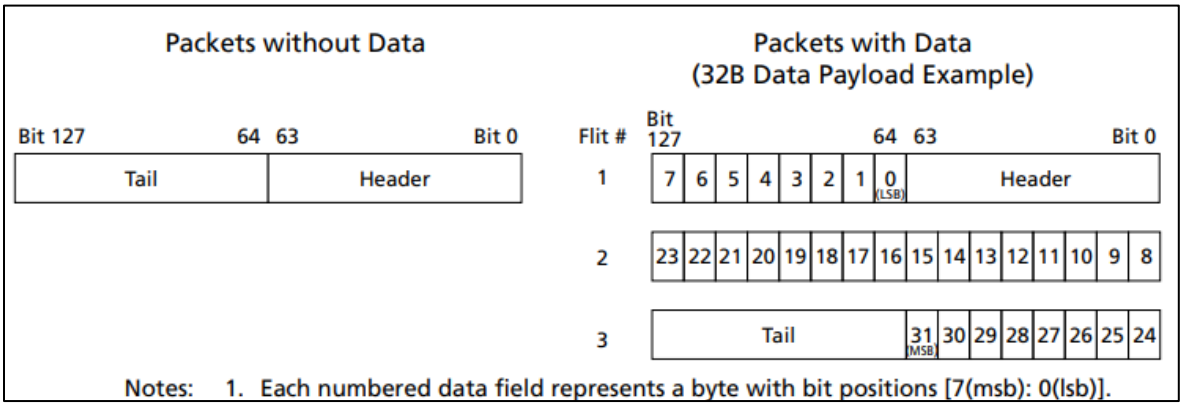


Figure 3.10: Packet layouts

A CRC field is included in the tail of every packet that covers the entire packet, including the header, all data, and the nonCRC tail bits. The link retry logic retransmits packets across the link if link errors are detected. Upon successful transmission of packets across the link, the packets are

transmitted through the logic base all the way to the destination vault controller. CRC provides command and data through-checking to the destination vault controller. If the vault controller detects a correctable error, error correction is executed before the data is returned. If an uncorrectable error is detected, an error status is returned in the response packet back to the requester. The CRC algorithm used on the HMC is the Koopman CRC-32K. This algorithm was chosen for the HMC because of its balance of coverage and ease of implementation. The polynomial for this algorithm is:

$$x^{32} + x^{30} + x^{29} + x^{28} + x^{26} + x^{20} + x^{19} + x^{17} + x^{16} + x^{15} + x^{11} + x^{10} + x^{7} + x^{6} + x^{4} + x^{2} + x + 1$$

The CRC calculation operates on the LSB of the packet first. The packet CRC calculation must insert 0s in place of the 32-bits representing the CRC field before generating or checking the CRC. For example, when generating CRC for a packet, bits [63: 32] of the Tail presented to the CRC generator should be all zeros. The output of the CRC generator will have a 32-bit CRC value that will then be inserted in bits [63:32] of the Tail before forwarding that FLIT of the packet. When checking CRC for a packet, the CRC field should be removed from bits [63:32] of the Tail and replaced with 32-bits of zeros, then presented to the CRC checker. The output of the CRC checker will have a 32-bit CRC value that can be compared with the CRC value that was removed from the tail. If the two compare, the CRC check indicates no bit failures within the packet. Overall packet lengths are from 1 to 9 FLITs. Write request packets and read response packets will contain from 16B to 128B of data between the header and the tail. All requests are operationally closed as follows: [5]

- Request packets are uniquely identified with a tag field embedded in the packet.
- All read request packets have a corresponding read response packet returned to the requesting link that includes the corresponding request tag and requested data.
- Normal write request packets are acknowledged either with an explicit write response packet that includes the write request's tag, or optionally with the write request tag embedded in the TGA field of a read response packet.
    - This positive acknowledgment provides an indication to the requesting processor that the request has been executed without error.
- As the HMC does not use the TAG field of posted write requests, the host can populate this field with any value.
- If unrecoverable errors occur in the request path, they are captured in the error status that is included in the response packet.
- If unrecoverable errors occur in the request path for posted write requests, an error response packet is returned to the requester indicating an error. Flow packets are not considered a request, they do not have a valid tag, and they do not have a corresponding response packet. The flow of request packets and response packets are managed with tokens. Flow packets have no flow control and can be sent at any time by the link master.

3.8.1 Packet Length

As noted previously, packets are always multiples of 16-byte FLITs. Within each packet header there is a length (LNG) field that indicates the total number of FLITs in the packet. A packet that contains no data FLITs would have LNG = 1; this represents the single FLIT holding the 8-byte header and 8-byte tail. When generating response packets, the necessary data size is determined from the CMD field in the corresponding request packet. Specific packet lengths are provided in tables below. A Duplicate Length (DLN) field is included within each packet header to provide additional packet integrity. The value of DLN must equal that of LNG within a packet. The DLN field is compared to the LNG field prior to the CRC being checked. Note that LNG enables the packet parser to determine the location of the CRC field (where a value of 1 points to the current FLIT). A mis compare of DLN and LNG is treated like a CRC check and will trigger the link retry mechanism. If LNG does not equal DLN when the packet is generated at the host, link retries will be unsuccessful and a fatal error will result. [5]

3.8.2 Packet Flow Control

Flow control occurs in both directions on each link. The flow of transaction layer packets is managed with token counts. Each token represents the storage to hold one FLIT (16 bytes). The link slave input buffer temporarily stores transaction layer packets as they are received from the link. (Note that flow packets are not stored in the input buffer, and therefore, are not subject to flow control.) The minimum size of this buffer must be equal to or greater than the number of FLITs in a single packet of the largest supported length, but in general can hold several of the largest packets to enable a constant flow across the link. The available space in this input buffer is represented in a token count register at the link master. This gives the link master knowledge of the available buffering at the other end of the link and the ability to evaluate whether there is enough buffer space to receive every FLIT of the next packet to be transmitted. The token count register is loaded during the initialization sequence with the maximum number of tokens representing the available buffer space at the link slave when it is empty. As the link master sends each transaction layer packet across the link, it must decrement the token count register by the number of FLITs in the transmitted packet. As the packet is received and stored in the input buffer, its FLITs use up the space represented by the decremented value of token count register at the link master. As each packet is read out of the input buffer, it is forwarded to a destination, and its FLIT location is freed up. This requires a mechanism to return packet tokens to the link transmitter, using the opposite link direction. The input buffer control logic sends a count (representing the number of FLITs that were read out of the input buffer when the packet was forwarded) to the local (adjacent) link master. This count is returned in the return token count (RTC) field of the next possible packet traveling in the opposite direction on the link. At the link slave, the RTC field is extracted from the incoming packet and sent back to the original link transmitter where its value is added to the current value of the token count register. If no transaction layer packet traffic is occurring in the return direction, the link master must create a flow packet known as a token return (TRET) to return the token. Appropriate sizing of the link slave input buffer requires weighing the trade-offs between latency, throughput, silicon real-estate, and routability. If there is a temporary

pause in the packet forwarding priority at the output of the input buffer, a packet might not be emptied from the input buffer. This would cause a pause in the return of the tokens. As long as this is infrequent and the input buffer is sized to accommodate it, the packet flow will not be disrupted unless the link is running at 100% busy. If the input buffer is empty, a specific implementation may choose to bypass the input buffer and forward a packet immediately to its destination. In this case, the tokens for the packet would be immediately returned to the link master. As noted previously, the link master must not transmit a packet if there is insufficient space in the input buffer at the other end of the link. The LNG field within the header of each outgoing packet must be compared to the token count register to determine whether the packet should be transmitted or the packet stream should be paused. [5]

3.8.3 Tagging

Operational closure for requests is accomplished using the tag fields in request and response packets. The tag value remains associated with the request until a response is received indicating that the request has been executed. Tags in READ requests are returned with the respective read data in the read response packet header. Write response tags are returned either within a write response packet, or optionally within the return tag A (TGA) field of any other response packet being returned to the same source link. If the feature is not supported, the corresponding bits within response packets will be specified as reserved. In the case of posted write requests, because no response is returned and the HMC does not use the TAG field of posted write requests, the host can populate this field with any value. Note that when multiple host links are connected to the HMC, each response packet will be returned on the same link as its associated request. This keeps the tag range independent for each host link. Tag fields in packets are nine bits long, which is enough space for 512 tags. All tags are available for use. Tag assignment and reassignment are managed by the host. There is no required algorithm to assign tag values to requests. HMC does not use the tag for internal control or identification, only for copying the tag from the request packet to the response packet. Tags are assigned by control logic at the host link master and must not be used in another request packet until a response tag with the same tag number is returned to that host link. Other host links will use the same tag range of 512 tags, but they are uniquely identified by their association with each different host link. Thus, the total maximum number of unique tags is 512 times the number of host links. For example, if four links are connected from an HMC to the host, there are 2048 usable tags for requests to the HMC. As an implementation example, a host control logic might decide to provide a time-out capability for every transaction to determine whether a request or response packet has been lost or corrupted, preventing the tag from being returned. The timer in this implementation example would consider link retry periods that occur to account for response packets that are delayed, but still returned. [5]

3.8.4 Packet Integrity

The integrity of the packet contents is maintained with the CRC field in the tail of every packet. CRC is used to detect failures that occur not only on transmission across the link, but along the entire path. Note that CRC may be regenerated along the path if flow control fields within the

header or tail change. CRC regeneration is done in an overlapped fashion with respect to a CRC check to ensure that no single point of failure will go undetected. There may be cases when the first FLITs of a packet are forwarded before the tail is received and the CRC is checked. This may occur in the HMC's link slave after a request packet crosses a link and is done to avoid adding latency in the packet path. If the packet is found to have a CRC error as it passes through the link slave, the packet is "poisoned," meaning that a destination will recognize it as nonfunctional and will not use it. The link slave poisons the packet by inverting a recalculated value of the CRC and inserting that into the tail in place of the errored CRC. Note that if a packet travels across a link after it is poisoned, a link master will still be able to embed flow control fields in the packet by recalculating the CRC with the embedded values, then inverting the recalculated CRC so that the poisoned state will be maintained. Because the flow control fields are valid in a poisoned packet, it is stored in the retry buffer. In this case, the link slave on the other end of the link will recognize the poisoned CRC and will still extract the flow control fields. Whenever a packet is poisoned, the CMD, ADRS, TAG, TGA, and ERRSTAT fields are not valid, but the flow control fields (FRP, RRP, RTC) are valid. [5]

3.8.5 Request Packets

Request packets carry request commands from the requester (host or HMC link configured as pass-thru link) to the responder (HMC link configured as host link). All request packets are subject to normal packet flow control. [5]

3.8.5.1 Request Header Layout

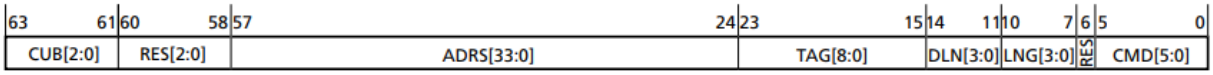


Figure 3.11: Request Packet Header Layout

Request packet headers for request commands and flow commands contain the fields shown in the table below. Specific commands may require some of the fields to be 0.

Table 3.12: Request Packet Header Fields

Name	Field Label	Bit Count	Bit Range	Function
Cube ID	CUB	3	[63:61]	CUB field used as an HMC identifier when multiple HMC devices are chained together. The HMC ignores bits in this field in non-chained topologies, except for including them in the CRC calculation.
Reserved	RES	3	[60:58]	Reserved: These bits are reserved for future address or Cube ID expansion. The responder will ignore bits in this field from the requester except for including them in the CRC calculation. The HMC can use portions of this field range internally.
Address	ADRS	34	[57:24]	Request address. For some commands, control fields are included within this range.
Tag	TAG	9	[23:15]	Tag number uniquely identifying this request.
Duplicate length	DLN	4	[14:11]	Duplicate of packet length field.
Packet length	LNG	4	[10:7]	Length of packet in FLITs (1 FLIT is 128 bits). Includes header, any data payload, and tail.
Reserved	RES	1	[6]	Reserved: The responder will ignore this bit from the requester except for including it in the CRC calculation. The HMC can use this field internally.
Command	CMD	6	[5:0]	Packet command.

3.8.5.2 Request Tail Layout

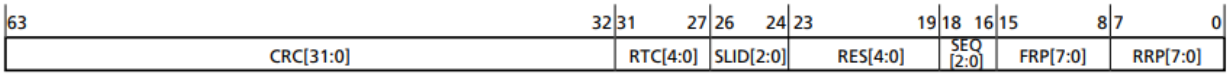


Figure 3.12: Request Packet Tail Layout

Table 3.13: Request Packet Tail Fields

Name	Field Label	Bit Count	Bit Range	Function
Cyclic redundancy check	CRC	32	[63:32]	The error-detecting code field that covers the entire packet.
Return token count	RTC	5	[31:27]	Return token count for transaction-layer flow control. In the request packet tail, the RTC contains the tokens that represent available space in the requester's input buffer.
Source Link ID	SLID	3	[26:24]	Used to identify the source link for response routing. The physical link number is inserted into this field by the HMC. The host can ignore these bits (except for including them in the CRC calculation) or alternatively, use the value in this field to validate the response packet is routed to the correct link source.
Reserved	RES	5	[23:19]	Reserved: The responder will ignore bits in this field from the requester except for including them in the CRC calculation. The HMC can use portions of this field range internally.
Sequence number	SEQ	3	[18:16]	Incrementing value for each packet transmitted, except for PRET and IRTRY packets.
Forward retry pointer	FRP	8	[15:8]	Retry pointer representing this packet's position in the retry buffer.
Return retry pointer	RRP	8	[7:0]	Retry pointer being returned for other side of link.

3.8.6 Response Packets

Response packets carry response commands from the responder (HMC link configured as host link) to the requester (host or HMC link configured as pass-thru link). All response packets are subject to normal packet flow control. [5]

3.8.6.1 Response Packet Header

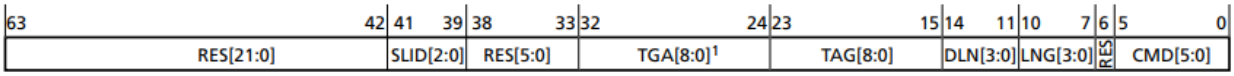


Figure 3.13: Response Packet Header Layout

Response packet headers for response commands contain the fields shown in the table below. Specific commands may require some of the fields to be 0.

Table 3.14: Response Packet Header Fields

Name	Field Label	Bit Count	Bit Range	Function
Reserved	RES	22	[63:42]	Reserved: The host will ignore bits in this field from the HMC except for including them in the CRC calculation.
Source Link ID	SLID	3	[41:39]	Used to identify the source link for response routing. The physical link number (0–3) is inserted into this field by the HMC. The host can ignore these bits (except for including them in the CRC calculation) or alternatively, use the value in this field to validate the response packet is routed to the correct link source
Reserved	RES	6	[38:33]	Reserved: The host will ignore bits in this field from the HMC except for including them in the CRC calculation.
Return tag (optional)	TGA	9	[32:24]	Field that can contain a write response tag
Tag	TAG	9	[23:15]	Tag number uniquely associating this response to a request.
Duplicate length	DLN	4	[14:11]	Duplicate of packet length field.
Packet length	LNG	4	[10:7]	Length of packet in FLITs (1 FLIT is 128 bits). Includes header, any data payload, and tail.
Reserved	RES	1	[6]	Reserved: The host will ignore this bit from the HMC except for including it in the CRC calculation.
Command	CMD	6	[5:0]	Packet command.

3.8.6.2 Response Packet Tail

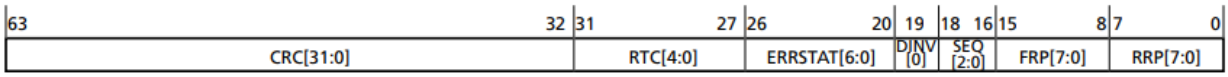


Figure 3.14: Response Packet Tail Layout

Table 3.15: Response Packet Tail Fields

Name	Field Label	Bit Count	Bit Range	Function
Cyclic redundancy check	CRC	32	[63:32]	The error-detecting code field that covers the entire packet.
Return token count	RTC	5	[31:27]	Return token count for transaction-layer flow control. In the response packet tail, the RTC contains the tokens that represent available space in the HMC input buffer.
Error status	ERRSTAT	7	[26:20]	Error status bits
Data Invalid	DINV	1	[19]	Indicates validity of packet payload. Data in packet is valid if DINV = 0 and invalid if DINV = 1.
Sequence number	SEQ	3	[18:16]	Incrementing value for each packet transmitted.
Forward retry pointer	FRP	8	[15:8]	Retry pointer representing this packet's position in the retry buffer.
Return retry pointer	RRP	8	[7:0]	Retry pointer being returned for other side of link.

The ERRSTAT field included in the response tail is valid for all read, write, and error responses. All bits in the ERRSTAT field being zero indicates a normal response with no errors or flags. There are 6 different error and flag categories defined by ERRSTAT[6:4] bits. The unique error or flag descriptions within each category indicate a specific error or item that requires attention. Table 3.16 shows the error categories and descriptions along with the method of response (read/write response packet or error response packet). [5]

Table 3.16: ERRSTAT[6:0] Bit Definitions

ERRSTAT Bit							Description	Response Packet Type
6	5	4	3	2	1	0		
0	0	0	0	0	0	0	No errors	
Link Errors								
0	1	0	0	0	0	0	Link 0 retry successful: Link 0 has successfully recovered from a link error by executing one or more link retries (within the retry attempt limit).	Error response packet (TAG = CUB number)
0	1	0	0	0	0	1	Link 1 retry successful: Link 1 has successfully recovered from a link error by executing one or more link retries (within the retry attempt limit).	
0	1	0	0	0	1	0	Link 2 retry successful: Link 2 has successfully recovered from a link error by executing one or more link retries (within the retry attempt limit).	
0	1	0	0	0	1	1	Link 3 retry successful: Link 3 has successfully recovered from a link error by executing one or more link retries (within the retry attempt limit).	
0	1	0	0	1	0	0	Link 4 retry successful: Link 4 has successfully recovered from a link error by executing one or more link retries (within the retry attempt limit).	
0	1	0	0	1	0	1	Link 5 retry successful: Link 5 has successfully recovered from a link error by executing one or more link retries (within the retry attempt limit).	
0	1	0	0	1	1	0	Link 6 retry successful: Link 6 has successfully recovered from a link error by executing one or more link retries (within the retry attempt limit).	

0	1	0	0	1	1	1	Link 7 retry successful: Link7 has successfully recovered from a link error by executing one or more link retries (within the retry attempt limit).	Error response packet (TAG = CUB number)
Protocol Errors								
ERRSTAT Bit							Description	Response Packet Type
6	5	4	3	2	1	0		
0	1	1	0	0	0	0	Invalid command: An unsupported command existed in a request packet.	Write response packet (TAG = tag of corresponding request)
0	1	1	0	0	0	1	Invalid length: An invalid length was specified for the given command in a request packet.	Read response packet (TAG = tag of corresponding request)
Fatal Errors								
1	1	1	0	0	0	0	Link 0 retry failed and was unsuccessful after the retry limit was reached.	Error response packet (TAG = CUB number)
1	1	1	0	0	0	1	Link 1 retry failed and was unsuccessful after the retry limit was reached.	
1	1	1	0	0	1	0	Link 2 retry failed and was unsuccessful after the retry limit was reached.	
1	1	1	0	0	1	1	Link 3 retry failed and was unsuccessful after the retry limit was reached.	
1	1	1	0	1	0	0	Link 4 retry failed and was unsuccessful after the retry limit was reached.	
1	1	1	0	1	0	1	Link 5 retry failed and was unsuccessful after the retry limit was reached.	
1	1	1	0	1	1	0	Link 6 retry failed and was unsuccessful after the retry limit was reached.	
1	1	1	0	1	1	1	Link 7 retry failed and was unsuccessful after the retry limit was reached.	

1	1	1	1	0	0	0	Input buffer overrun: The host has issued too many FLITs and has overrun the link input buffer, or has issued a mode command before the previous mode command has returned a response packet.	
1	1	1	1	0	1	1	Recoverable fatal error: HMC has identified a fatal error that will be repaired or recovered after a reset is performed.	

3.8.7 Flow Packets

Flow packets are generated at the link master to convey information for link flow control and link retry control to the link slave on the other end of the link. They are not part of the transaction layer. The link slave extracts the control information from the flow packets and then removes the flow packets from the packet stream without forwarding them to the link input buffer. For this reason, flow packets are not subject to normal packet flow control. The link master can generate and transmit flow packets without requiring tokens, and it will not decrement the token count when it transmits flow packets. Flow packets use the request packet header and tail layouts described in 2.8.5 “Request Packets”. The flow packet commands are described below in “Flow Commands”. Some header fields and tail fields within the flow packets are not used and should be 0. [5]

3.8.8 Request Commands

All request commands are issued by the requester (host or HMC link configured as pass-thru link) to the responder (HMC link configured as host link), using request packets described in 2.8.5 “Request Packets”. Every request must have a valid tag included in the request packet header. A corresponding response packet (one that includes the same tag) may be issued by the responder unless it is impossible for the responder to determine the correct tag number of the request due to a request packet CRC error. All request packets are saved in the link retry buffer as they cross a link. This means that these packets are retried if an error occurs on their transfer. [5]

Table 3.17: Transaction Layer – Request Commands

Command Description	Symbol	CMD Bit						Packet Length in FLITs	Corresponding Response Returned
		5	4	3	2	1	0		
WRITE Requests									
16-byte WRITE request	WR16	0	0	1	0	0	0	2	WR_RS
32-byte WRITE request	WR32	0	0	1	0	0	1	3	WR_RS
48-byte WRITE request	WR48	0	0	1	0	1	0	4	WR_RS
64-byte WRITE request	WR64	0	0	1	0	1	1	5	WR_RS
80-byte WRITE request	WR80	0	0	1	1	0	0	6	WR_RS
96-byte WRITE request	WR96	0	0	1	1	0	1	7	WR_RS
112-byte WRITE request	WR112	0	0	1	1	1	0	8	WR_RS
128-byte WRITE request	WR128	0	0	1	1	1	1	9	WR_RS
MODE WRITE, BIT WRITE, and ATOMIC Requests									
MODE WRITE request	MD_WR	0	1	0	0	0	0	2	MD_WR_RS
BIT WRITE request	BWR	0	1	0	0	0	1	2	WR_RS
Dual 8-byte add immediate	2ADD8	0	1	0	0	1	0	2	WR_RS
Single 16-byte add immediate	ADD16	0	1	0	0	1	1	2	WR_RS
POSTED WRITE Requests									
16-byte POSTED WRITE request	P_WR16	0	1	1	0	0	0	2	None
32-byte POSTED WRITE request	P_WR32	0	1	1	0	0	1	3	None
48-byte POSTED WRITE request	P_WR48	0	1	1	0	1	0	4	None
64-byte POSTED WRITE request	P_WR64	0	1	1	0	1	1	5	None
80-byte POSTED WRITE request	P_WR80	0	1	1	1	0	0	6	None
96-byte POSTED WRITE request	P_WR96	0	1	1	1	0	1	7	None
112-byte POSTED WRITE request	P_WR112	0	1	1	1	1	0	8	None

128-byte POSTED WRITE request	P_WR128	0	1	1	1	1	1	9	None
POSTED BIT WRITE, and POSTED ATOMIC Requests									
Posted BIT WRITE request	P_BWR	1	0	0	0	0	1	2	None
Posted dual 8-byte add immediate	P_2ADD8	1	0	0	0	1	0	2	None
Posted single 16-byte add immediate	P_ADD16	1	0	0	0	1	1	2	None
Read Requests									
16-byte READ request	RD16	1	1	0	0	0	0	1	RD_RS
32-byte READ request	RD32	1	1	0	0	0	1	1	RD_RS
48-byte READ request	RD48	1	1	0	0	1	0	1	RD_RS
64-byte READ request	RD64	1	1	0	0	1	1	1	RD_RS
80-byte READ request	RD80	1	1	0	1	0	0	1	RD_RS
96-byte READ request	RD96	1	1	0	1	0	1	1	RD_RS
112-byte READ request	RD112	1	1	0	1	1	0	1	RD_RS
128-byte READ request	RD128	1	1	0	1	1	1	1	RD_RS
Mode Read Requests									
MODE READ request	MD_RD	1	0	1	0	0	0	1	MD_RD_RS

3.8.8.1 READ and WRITE Request Commands

READ and WRITE request commands are issued by the requester to access the DRAM memory. They are block commands that access a contiguous number of memory-addressable bytes with 16-byte granularity, meaning they can begin and end on any 16- byte boundary up to a maximum length of 128 bytes. Due to the 16-byte granularity of READ and WRITE request commands, the four LSBs of the request address are ignored. There are unique commands for each supported memory access length. READ and WRITE operations will not cross a row (page) boundary within the DRAM array. The maximum block size selected in the address map mode register determines the boundary at which the addressing will wrap if the block access goes beyond the end of the maximum block size. All read requests are acknowledged with a read response packet that includes the tag of the request and the data payload. All write requests (nonposted) are acknowledged either

with an explicit write response packet that includes the tag of the write request or, if supported, with the write request tag embedded in the TGA field of some other response packet. If a READ or WRITE request command specifies a data payload length that is longer than the configured maximum block size, it will be treated as an invalid command. [5]

3.8.8.2 POSTED WRITE Request Commands

POSTED WRITE request commands operate similarly to standard write requests, except that there is no response returned to the requester. As the HMC does not use the TAG field of posted write requests, the host can populate this field with any value. If an error occurs during the execution of the write request, an Error Response packet will be generated indicating the occurrence of the error to the host. Note that in this case, the write request's tag will not be included in the Error Response packet. [5]

3.8.8.3 ATOMIC Request Commands

Atomic requests involve reading 16 bytes of data from DRAM (as accessed by the request address field), performing an operation on the data through the use of a 16-byte operand (also included in request packet), and then writing the results back to the same location in DRAM. The read-update-write sequence occurs atomically, meaning that no other request can access the same bank until the write of the atomic request is complete. Atomic requests are similar to a 16-byte write request in that they have a 16-byte data payload in the request packet, and a write response may or may not be returned (dependent on posted or nonposted request). The data resulting from the ATOMIC operation is not returned in a response command. The following two sections describe the currently defined ATOMIC operations supported by the HMC. [5]

1) Dual 8-Byte Add Immediate:

This atomic request performs two parallel ADD IMMEDIATE operations. Each add operation uses an 8-byte memory operand from DRAM and a 4-byte two's complement signed integer immediate operand from the atomic request data payload. If the result exceeds 263 the carry-out of the most significant bit is dropped and the result rolls over. The results are written back to DRAM, overwriting the original memory operands. [5]

Table 3.18: Operands from DRAM

Byte	15 (MSB)	14	13	12	11	10	9	8 (LSB)	7 (MSB)	6	5	4	3	2	1	0 (LSB)
Field	Memory Operand 2								Memory Operand 1							

Table 3.19: Immediate Operands Included in Atomic Request Data Payload

Byte	15	14	13	12	11 (MSB)	10	9	8 (LSB)	7	6	5	4	3 (MSB)	2	1	0 (LSB)
Field	4 bytes of zeros				Imm operand 2				4 bytes of zeros				Imm operand 1			

2) Single 16-Byte Add Immediate:

This request performs a single ADD IMMEDIATE operation. It uses a 16-byte memory operand from DRAM and an 8-byte two's complement signed integer immediate operand from the atomic request data payload. If the results exceed 2127 the carry-out of the most significant bit is dropped and the result rolls over. The result is written back to the DRAM, overwriting the original memory operand. [5]

Table 3.20: Operand from DRAM

Byte	15 (MSB)	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0 (LSB)
Field	Memory operand 1															

Table 3.21: Immediate Operand Included in Atomic Request Data Payload

Byte	15	14	13	12	11	10	9	8	7 (MSB)	6	5	4	3	2	1	0 (LSB)
Field	8 bytes of zeros								Immediate Operand 1							

3.8.8.4 MODE READ and WRITE Request Commands

MODE READ and WRITE request commands access the internal hardware mode and status registers within the logic layer of the HMC. Each mode request accesses up to 32 contiguous bits of the register. The address field within the mode request packet is redefined to provide a register address, a start field, and a size field as shown in the table below. MODE READ and WRITE request commands must include address values that point to the registers defined. A MODE READ request command that contains an address that points to a non-existent register will get a MODE READ response back that includes all 0's within the data payload. A MODE WRITE request that points to a non-existent register will not be performed. [5]

Table 3.22: Mode Request Addressing

Field	Bits	Number of Bits	Description
Unused	33:32	2	Unused bits
Start	31:27	5	Start field: Encoded to provide a value from 0 to 31 that points to the starting bit position within the 32 bits of data, 0 being the LSB position and 31 being the most significant bit (MSB) position.
Size	26:22	5	Size field: Indicates the number of contiguous bits to read or write (from 1 to 32), and is encoded as follows: 0x0 = 32 bits, 0x1–0x1F = 1–31 bits. This implies that the minimum size is 1 bit
Register address	21:0	22	Register address field: Points to a specific mode or status register. All registers are 32 bits or less in size.

Notes:
A full 32-bit access would be 0 in both the start and size fields.
The valid data bytes within the data payload in the mode write request packet and the mode read response packet are right-justified at the four least significant bytes as shown in the following table.

Table 3.23: Valid Data Bytes

Byte	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	Unused												Data			

If less than 32 bits are being accessed, as determined by the start and size fields within the address, the valid data bits are right-justified to the LSBs within the 4-byte data field. The requester can only issue one mode command at a time and must wait for the corresponding response before issuing another mode command. Because of this requirement, the HMC does not support posted MODE WRITE requests.

3.8.8.5 BIT WRITE Command

The BIT WRITE request is a 2-FLIT request packet that provides a 0- to 8-byte write capability as indicated by write data mask bits included in the data payload. The three LSBs of the byte address field within the address in the request packet header must be 000 when addressing an 8-byte block. The specific bits to be written within the 8-byte block are identified by the data mask bits in bytes [15:8] of the 16-byte data payload within the request packet as shown in the following table. [5]

Table 3.24: Request Packet Bytes to be written

Byte	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	Data mask								Write Data							

Notes:

1. Data mask: Each bit inhibits writing the corresponding write data bit when 1.

2. Write data: Up to eight bytes of byte-aligned write data. The bit(s) to be written must be positioned in the correct bit position within the write data field, as identified with the cleared bit(s) in the data mask field. For example, if the data mask = 0x00FFFFFFFFFF00FF, then bytes 7 and 1 of the write data field will be written into the same bytes in the memory. If the data mask = 0xFFFFFFFFFFFFFFFF, the READ-UPDATE-WRITE operation will be performed with the original value of the data being rewritten into the memory.

3.8.9 Response Commands

All response commands are issued by the responder (HMC link configured as host link) to the requester (host or HMC link configured as pass-thru link), using response packets described in 2.8.6 “Response Packets”. Every response other than the error response includes a valid tag that matches the tag in its corresponding request. All response packets are saved in the link retry buffer as they cross a link, which means they are retried if an error occurs on the link. [5]

Table 3.25: Transaction Layer – Response Commands

Command Description	Symbol	CMD Bit						Packet Length in FLITS
		5	4	3	2	1	0	
READ response	RD_RS	1	1	1	0	0	0	1 + data FLITs
WRITE response	WR_RS	1	1	1	0	0	1	1
MODE READ response	MD_RD_RS	1	1	1	0	1	0	2
MODE WRITE response	MD_WR_RS	1	1	1	0	1	1	1
ERROR response	ERROR	1	1	1	1	1	0	1
Reserved	-	1	1	1	1	1	1	-

3.8.9.1 READ and MODE READ Response Commands

A READ or MODE READ response command is issued by the responder to return requested data to the requester. A read response packet always includes data payload along with the header and tail, and it includes the same tag as its corresponding read request. If an error occurred such that the data is not valid, as indicated in the DINV field in the response tail, the data payload FLITs are still present in the response packet but they hold invalid data. For this reason, the response packet lengths are always consistent with the corresponding request. If supported, a read response packet could also include a tag from a write request embedded in the TGA field of the response packet tail. This acts as a write response, providing a positive acknowledgment of the completion of a successful write request with no errors. The write response tag will always be returned to the same link where the original write request was received. A MODE READ command must include address values that point to the registers defined in Section 10 in the specs document [5]. A MODE READ request command that contains an address that points to a non-existent register will get a MODE READ response back that includes all 0s within the data payload, and the DINV bit will not be set. [5]

3.8.9.2 WRITE and MODE WRITE Response Commands

The WRITE or MODE WRITE response command provides an explicit write response packet that returns the tag for the original write request. If errors occurred during the transfer or execution of the WRITE request command, status will be included in the ERRSTAT field of the write response packet. A lack of error status bits indicates that the write was successful. If supported, any write response tag that is included in a TGA field of another response indicates that there are no associated errors to be reported for the corresponding write request and that the write was successful. Mode write responses cannot be embedded within the TGA field and are sent through the MODE WRITE response command only. A MODE WRITE command must include address values that point to the registers. A MODE WRITE request command that contains an address that points to a non-existent register will get a MODE WRITE response back, however the request will not be performed. [5]

3.8.9.3 ERROR Response Command

The HMC uses the READ, MODE READ, WRITE, and MODE WRITE response commands to report errors or flags that are associated with a specific request and tag. These error reports are included in the ERRSTAT field of the response packet. The ERROR response command is used by the HMC to report a critical event when the request type and/or the original request tag are not known. The three LSBs of the ERROR response command packet tag field contain the values the CUB[2:0] signals are tied to. TAG[2:0] = CUB[2:0] and TAG[8:3] = 0. This gives the host visibility to which HMC the Error Response packet originated from in a topology with multiple HMCs. The ERROR response command is unsolicited and can be configured to transmit ERROR response commands on any or all links. The following situations cause the HMC to transmit an ERROR response command packet: [5]

- The responder receives a request packet from the requester that is successfully transferred across a link but is internally corrupted before it reaches its destination. In this case the original command and tag are lost and the responder cannot generate a proper response packet or the necessary tag to close the transaction.
- The HMC has an internal error not associated with a transaction. • The HMC reports a condition or status to the host.

3.8.10 Flow Commands

Flow commands facilitate retry and flow control on the link and are not part of the transaction layer. They are generated at the link master to send information to the other end of the link when no other link traffic is occurring or when a link retry sequence is to be initiated. Flow commands are not forwarded from that point. Flow commands use the request packet header and tail layouts described in "Request Packets" section, but are not considered requests, and do not have corresponding response commands. Flow commands are not subject to token flow control. The link master is allowed to send a flow command even if there are no tokens available. [5]

Table 3.26: Flow Commands

Command Description	Symbol	CMD Bit						Packet Length in FLITS
		5	4	3	2	1	0	
Null	NULL	0	0	0	0	0	0	NA
Retry pointer return	PRET	0	0	0	0	0	1	1
Token return	PRET	0	0	0	0	1	0	1
Init retry	IRTRY	0	0	0	0	1	1	1

3.8.10.1 NULL Command

A NULL command field of all zeros, along with all zeros in the rest of the FLIT, define a NULL FLIT. NULL FLITs are driven on the link when there are no packets to be transmitted. (Null FLITs are also used during link initialization.) Note that a FLIT of all zeros will have a correct CRC. A poisoned version of the NULL FLIT is not supported. [5]

3.8.10.2 RETRY POINTER RETURN (PRET) Command

This command is issued by the link master to return retry pointers when there is no other link traffic flowing at the time the pointer is to be returned. It is a single-FLIT packet with no data payload. PRET packets are not saved in the retry buffer, so their SEQ and FRP fields should be set to 0. Tokens should not be returned in these packets, so the RTC field should be set to 0. [5]

3.8.10.3 TOKEN RETURN (TRET) Command

This command is issued by the link master to return tokens when there is no other link traffic flowing at the time the token is to be returned. It is a single-FLIT packet with no data payload. Token return packets are saved in the retry buffer, so their SEQ and FRP fields are valid. [5]

3.8.10.4 INIT RETRY (IRTRY) Command

A programmable stream of INIT RETRY (IRTRY) command packets is issued by the link master when it is in the Retry_Init state. This is a single-FLIT packet with no data payload. IRTRY command packets are not saved in the retry buffer, so their SEQ field should be set to 0. Tokens should not be returned in these packets, so the RTC field should also be set to 0. [5]

3.8.11 Valid Field Command Summary

Table 3.27: Valid Field and Command Summary Table

Symbol	Request		Response			Flow			
	Read	Write	Read	Write	Error	NULL	PRET	TRET	IRTRY
ADRS	V	V	N/A	N/A	N/A	0	0	0	0
CMD	V	V	V	V	V	0	V	V	V
CRC	V	V	V	V	V	0	V	V	V
DLN	V	V	V	V	V	0	V	V	V
ERRSTAT	N/A	N/A	V	V	V	N/A	N/A	N/A	N/A
FRP	V	V	V	V	V	0	0	V	V
LNG	V	V	V	V	V	0	V	V	V
RRP	V	V	V	V	V	0	V	V	V
RTC	V	V	V	V	V	0	0	V	0
SEQ	V	V	V	V	V	0	0	V	0
TAG	V	V	V	V	V	0	0	0	0
TGA (optional)	N/A	N/A	V	V	V	N/A	N/A	N/A	N/A

3.8.12 Transaction Layer Initialization

During link initialization, after the responder stops sending TS1 patterns on the link, the link layer is considered active. Both ends of the link are in the active state, sending NULL FLITs. At this

point, the HMC will continue internal initialization. Transaction layer initialization continues with the following steps:

1. After internal initialization is complete, the responder will send one or more TRET packets that will transfer the number of its link input buffer tokens to the requester. Note that each TRET packet can transmit a count of 31 tokens, therefore there will be multiple TRETs required to transfer the entire number of tokens representing the link input buffer's available space.
2. Once the requester receives at least one TRET packet from the responder, the requester should transmit a series of TRET packets back to the responder carrying the tokens representing the available space in the requester's link input buffer. The HMC can support up to 1023 tokens from the host on each link.
3. After the TRET packets are transmitted from the requester to the responder, the requester can now start sending transaction packets. There is no required timeframe when the transaction packets can start.

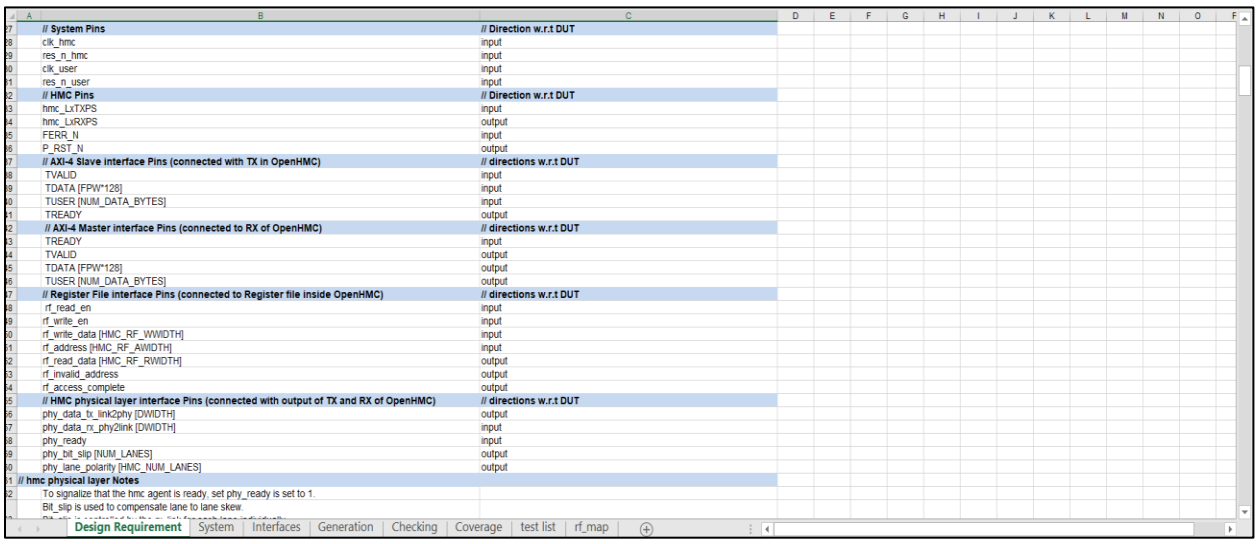
Within an HMC, all links that are configured as pass-thru links must complete their transaction layer initialization before the links configured as host links begin to send TRET packets upstream. This ensures that all links within a multi-cube topology are initialized once the link configured as a host link in source mode has finished its transaction layer initialization. [5]

Chapter 4
Verification Plan & Features

We've made a verification plan based on the design specs and architecture mentioned in the previous chapter, this plan is an Excel sheets file which has 8 sheets which are:

4.1 Design Requirements Sheet

In this sheet we have included the features of the design specs mentioned in the previous chapter each feature in a cell.



// System Pins	**// Direction w.r.t DUT**
clk_hmc	input
res_n_hmc	input
clk_user	input
res_n_user	input
// HMC Pins	**// Direction w.r.t DUT**
hmc_LxTXPS	input
hmc_LxRXPS	output
FERR_N	input
P_RST_N	output
// AXI-4 Slave interface Pins (connected with TX in OpenHMC)	**// directions w.r.t DUT**
TVALID	input
TDATA [FPW*128]	input
TUSER [NUM_DATA_BYTES]	input
TREADY	output
// AXI-4 Master interface Pins (connected to RX of OpenHMC)	**// directions w.r.t DUT**
TREADY	input
TVALID	output
TDATA [FPW*128]	output
TUSER [NUM_DATA_BYTES]	output
// Register File interface Pins (connected to Register file inside OpenHMC)	**// directions w.r.t DUT**
rf_read_en	input
rf_write_en	input
rf_write_data [HMC_RF_WWIDTH]	input
rf_address [HMC_RF_AWIDTH]	input
rf_read_data [HMC_RF_RWIDTH]	output
rf_invalid_addtress	output
rf_access_complete	output
// HMC physical layer interface Pins (connected with output of TX and RX of OpenHMC)	**// directions w.r.t DUT**
phy_data_tx_link2phy [DWIDTH]	output
phy_data_rx_phy2link [DWIDTH]	input
phy_ready	input
phy_bit_slip [NUM_LANES]	output
phy_lane_polarity [HMC_NUM_LANES]	output
// hmc physical layer Notes	
To signalize that the hmc agent is ready, set phy_ready is set to 1.	
Bit_slip is used to compensate lane to lane skew.	

Design Requirement | System | Interfaces | Generation | Checking | Coverage | test list | rf_map

Figure 4.1: Design Requirements Sheet

4.2 System Sheet

The same as the Design Requirements Sheet but with an added column called “Approach” which specifies how features will be treated or implemented in the testbench and how the testbench will be adapted to have full control on these features.

	A	B		C	D
1		OpenHMC			
2		Design Requirement Description			Approach
28		clk_hmc		input	clock will be generated at testbench
29		res_n_hmc		input	module
30		clk_user		input	for simplicity make clk_user =
31		res_n_user		input	clk_hmc
32	// Request Packets				
33		Request packets are issued by the requester (host or HMC configured as a pass-thru link). The request packet header includes address information to perform a READ or WRITE operation.			the sequence item hmc_packet_seq_item should contain all the header and tail fields
34		// Request Packets - Header Fields (64 bits)			
		// bit count	// bit range in header	// Description	
35		CUB	3	[63:61]	Cube ID field used to match request with target cube. The HMC ignores bits in this field in non-chained topologies, except for including them in the CRC calculation.	keep at zero
36		RES	3	[60:58]	Reserved: These bits are reserved for future address or Cube ID expansion. The responder will ignore bits in this field from the requester except for including them in the CRC calculation. The HMC can use portions of this field range internally	
37		ADRS	34	[57:24]	Request address. For some commands, control fields are included within this range	
38		TAG	9	[23:15]	Tag number uniquely identifying this request	
39		DLN	4	[14:11]	Duplicate of packet length field.	
40		LNG	4	[10:7]	Packet length in FLITs (1 FLIT is 128 bits). Includes header, data payload & tail	will be constrained for command type
41		RES	1	[6]	Reserved	
42		CMD	6	[5:0]	Packet Command	will be of enumeration type
43		// Request Packets - Tail Fields (64 bits)			
		// bit count	// bit range in tail	// Description	
44		CRC	32	[63:32]	Cyclic redundancy check. The error-detecting code field that covers the entire packet	after we generate the request packet with zero tail, will send it to scoreboard where it will calculate the CRC, then compare with the output of the DUT
45		RTC	5	[31:27]	Return token count for transaction-layer flow control. In the request packet tail, the RTC contains the encoded value for tokens that represent available space in the requester's input buffer	
		SLID	3	[26:24]	Source Link ID. Used to identify the source link for response routing. The physical link number is inserted into this field by the HMC. The	keep at zero

Design Requirement | System | Interfaces | Generation | Checking | Coverage | test list | rf_map

Figure 4.2: System Sheet

4.3 Interfaces Sheet

This sheet contains the design interfaces and pins and has a column called “Approach” used to specify how the testbench should deal with these interfaces and pins. Also, it has another column called “component name” which specifies the components that would be implemented in the testbench. There is also a column for notes about these components.

	A	B	C	D	E	F
1		OpenHMC				
2		Design Requirement Description		Approach	component name	Notes
3		# AXI-4 Slave interface Pins (connected with TX in OpenHMC)	# directions w.r.t DUT			
4		TVALID	input	implement AXI RX agent to monitor and drive this pins	axi_tx_agent	master agent
5		TDATA [FPW*128]	input		axi_tx_driver	
6		TUSER [NUM_DATA_BYTES]	input		axi_tx_monitor	
7		TREADY	output		axi_tx_sequencer	
8		# AXI-4 Master interface Pins (connected to RX of OpenHMC)	# directions w.r.t DUT			
9		TREADY	input	implement AXI RX agent to monitor this pins	axi_rx_agent	passive agent
10		TVALID	output	bfm of axi agent will handle the TREADY pin	axi_rx_monitor	
11		TDATA [FPW*128]	output		axi_rx_config	
12		TUSER [NUM_DATA_BYTES]	output			
13	# AXI-4 Interface Notes					
14		1. TREADY - TX: openHMC is ready to sample TDATA and TUSER - RX: user is ready to sample TDATA and TUSER ■ TDATA and TUSER will not change when TREADY=0		in axi_bfm wait for both TVALID and TREADY to be asserted to pass the data between OpenHMC and axi agents		
15		2. TVALID - TX: TDATA and TUSER are sampled on TX when TVALID and TREADY = 1 - RX: TDATA and TUSER are valid when TVALID=1 ■ TVALID may be held high even when TREADY = 0 ■ TREADY may be held high even when TVALID=0.		- no restrictions on when TVALID and TREADY are asserted		
16		3. TDATA [FPW*128] - The TDATA bus expects complete HMC request packets, starting with the 64bit header followed by data FLITs. - a single AXI cycle can carry (parts of) multiple packets on both interfaces, TX and RX. - The user is responsible to populate all request header fields and The tail must be set to all zeroes ■ TVALID must be held high until the entire packet (including its tail) was transmitted - On RX, the openHMC controller outputs complete HMC response packets				
17		4. TUSER [NUM_DATA_BYTES] - NOTE: 3*FPW on TX and 4*FPW on RX - valid [FPW-1:0]: valid FLIT indicator, one bit per FLIT in 1 axi cycle - hdr [(2*FPW)-1:FPW]: header indicator, if the FLIT includes a header - tail [(3*FPW)-1:2*FPW]: tail indicator, if the FLIT includes a tail - only in RX, err [4*FPW-1:3*FPW]: error response packet indicator				
18		■ For proper operation of the interface, all FLITs of a packet on TX must be shifted in continuously without any 'bubble'/NULL FLITs or cycles in between.		don't leave any free slots in the axi cylce		
19		■ the frequency of the user clock clk_user driving the AXI-4 interface must be equal to or higher than clk_hmc				
20		# Register File interface Pins (connected to Register file inside OpenHMC)	# directions w.r.t DUT			
21		rf_read_en	input	implement Register File agent to monitor and drive this pins	rf_agent	register model
22		rf_write_en	input		rf_driver	
23		rf_write_data [HMC_RF_WWIDTH]	input		rf_monitor	
24		rf_address [HMC_RF_AWIDTH]	input		rf_config	
25		rf_read_data [HMC_RF_RWIDTH]	output		rf_sequencer	
26		rf_invalid_address	output			
27		rf_access_complete	output			

Design Requirement | System | Interfaces | Generation | Checking | Coverage | test list | rf_map

Figure 4.3: Interfaces Sheet

4.4 Generation Sheet

This sheet contains the sequence items and sequences that would be implemented in the testbench corresponding to each interface feature in the design specs. This sheet has 3 added columns which are “Generation”, “component name” and “extends”. The “Generation” column specifies the function of the sequence item or the sequence that would be generated at each interface feature. The “component name” column specifies the name of the sequence item or the sequence that would be generated at each interface feature. Finally, the “extends” column specifies what type the component generated at each interface feature is, whether extends sequence or sequence item.

	Design Requirement Description	Generation	component name	extends
	OpenHMC			
3	# AXI-4 Interface Notes			
4	1. TREADY - TX: openHMC is ready to sample TDATA and TUSER - RX: user is ready to sample TDATA and TUSER ■ TDATA and TUSER will not change when TREADY=0			
5	2. TVALID - TX: TDATA and TUSER are sampled on TX when TVALID and TREADY = 1 - RX: TDATA and TUSER are valid when TVALID=1 ■ TVALID may be held high even when TREADY = 0 ■ TREADY may be held high even when TVALID=0.			
6	3. TDATA [FPW*128] - The TDATA bus expects complete HMC request packets, starting with the 64bit header followed by data FLITs - a single AXI cycle can carry (parts of) multiple packets on both interfaces, TX and RX - The user is responsible to populate all request header fields and The tail must be set to all zeroes ■ TVALID must be held high until the entire packet (including its tail) was transmitted - On RX, the openHMC controller outputs complete HMC response packets	- create a sequence item for the request packets, the generated hmc_packet_seq_item will have a constraint on the tail to be all zeros	hmc_packet_seq_item axi_request_packet_sequence	sequence_item sequence
7	4. TUSER [NUM_DATA_BYTES] - NOTE: 3*FPW on TX and 4*FPW on RX - valid [FPW-1:0] : valid FLIT indicator, one bit per FLIT in 1 axi cycle - hdr [(2*FPW)-1:FPW] : header indicator, if the FLIT includes a header - tail [(3*FPW)-1:2*FPW] : tail indicator, if the FLIT includes a tail - only in RX, err [rspB(4*FPW)-1:3*FPW] : error response packet indicator			
8	■ For proper operation of the interface, all FLITs of a packet on TX must be shifted in continuously without any “bubble”/NULL FLITs or cycles in between.	don't leave any free slots in the axi cylce		
9	■ the frequency of the user clock clk_user driving the AXI-4 interface must be equal to or higher than clk_hmc			
10	# Register File interface Pins (connected to Register file inside OpenHMC)	# directions w.r.t DUT		
11	rf_read_en	input		
12	rf_write_en	input		
13	rf_write_data [HMC_RF_WWIDTH]	input		
14	rf_address [HMC_RF_AWIDTH]	input		
15	rf_read_data [HMC_RF_RWIDTH]	output		
16	rf_invalid_address	output		
17	rf_access_complete	output		
18	# Register File Interface Notes			
19	A Register File module allows to control and monitor the openHMC operation.	create a sequence item contains data addr write while a rf_bfm will handle the enable signals	rf_seq_item	sequence_item
20	the target address must be applied before asserting read_en or write_en			
	Write Operation - write_data must hold the 64-bit value to be written - Data is sampled when write_enable is asserted	create a sequence for write operation: - set address - set data	rf_write_sequence	sequence

Figure 4.4: Generation Sheet

4.5 Checking Sheet

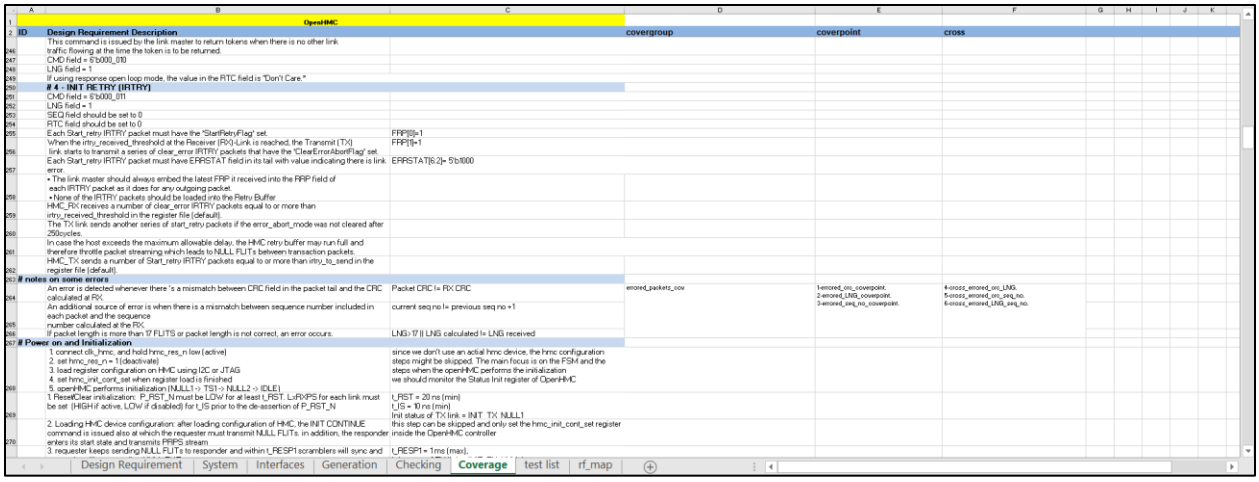
This sheet has 4 added columns which are “Approach”, “check”, “check type” and “comment”. The “Approach” column specifies what happens at each interface feature to be checked. The “check” column specifies what checks should be done at these features. The “check type” column specifies the type of check that should be done at these features, whether it is assertion or done through scoreboard. Finally, the “comment” column mentions the order number of each check.

	ID	Design Requirement Description	Approach	check	check type	Comment
		OpenHMC				
14	20	phy_lane_polarity is used to connect transciever polarity inputs if polarity is controlled within the transceivers and “CTRL_LANE_POLARITY” is set to 1.				
15	# AXI-4 Interface Notes					
16	29	1. TREADY - TX: openHMC is ready to sample TDATA and TUSER - RX: user is ready to sample TDATA and TUSER ■ TDATA and TUSER will not change when TREADY=0				
17	30	2. TVALID - TX: TDATA and TUSER are sampled on TX when TVALID and TREADY = 1 - RX: TDATA and TUSER are valid when TVALID=1 ■ TVALID may be held high even when TREADY = 0 ■ TREADY may be held high even when TVALID=0.				
18	31	3. TDATA [FPW*128] - The TDATA bus expects complete HMC request packets, starting with the 64bit header followed by data FLITs - a single AXI cycle can carry (parts of) multiple packets on both interfaces, TX and RX - The user is responsible to populate all request header fields and The tail must be set to all zeroes ■ TVALID must be held high until the entire packet (including its tail) was transmitted - On RX, the openHMC controller outputs complete HMC response packets	tdata doesn't change while tready not asserted	tdata_dont_change_while_tready_not_asserted	assertion	check#1
19	32	4. TUSER [NUM_DATA_BYTES] - NOTE: 3*FPW on TX and 4*FPW on RX - valid [FPW-1:0] : valid FLIT indicator, one bit per FLIT in 1 axi cycle - hdr [(2*FPW)-1:FPW] : header indicator, if the FLIT includes a header - tail [(3*FPW)-1:2*FPW] : tail indicator, if the FLIT includes a tail - only in RX, err [rspB(4*FPW)-1:3*FPW] : error response packet indicator	tuser doesn't change while tready not asserted	tuser_dont_change_while_tready_not_asserted	assertion	check#2
20	33	■ For proper operation of the interface, all FLITs of a packet on TX must be shifted in continuously				
21	34	without any “bubble”/NULL FLITs or cycles in between.				
22	35	■ the frequency of the user clock clk_user driving the AXI-4 interface must be equal to or higher than clk_hmc				
23	# Register File Interface Notes					
24	45	A Register File module allows to control and monitor the openHMC operation.				
25	46	the target address must be applied before asserting read_en or write_en	make sure to assign address first			
26	47	Write Operation - write_data must hold the 64-bit value to be written - Data is sampled when write_enable is asserted				
27	48	Read Operation - read enable must be asserted - read data contained the monitored value				
28	49	■ Each operation is confirmed by the access_complete signal set for one cycle	data is valid after access_complete is asserted			
29	50	In case that an invalid address was applied, invalid_address will remain as long as read_en or write_en are active	make an assertion that invalid_address never asserted	invalid_address_not_asserted	assertion	check#3
30	51	■ The user must not assert write_en and read_en both at the same time.	make an assertion that both won't be 1 at the same time	enable_not_asserted_together	assertion	check#4
31		# Register File Address Map				

Figure 4.5: Checking Sheet

4.6 Coverage Sheet

This sheet has 3 added columns which are “covergroup”, “coverpoint” and “cross”. The “covergroup” column specifies the cover group that should be covered at each interface feature. The “coverpoint” column specifies the points in each cover group that should be covered at each interface feature. Finally, the “cross” column specifies the crossed cover points (two or more cover points that are mixed together to allow wider coverage) in each cover group that should be covered at each interface feature.



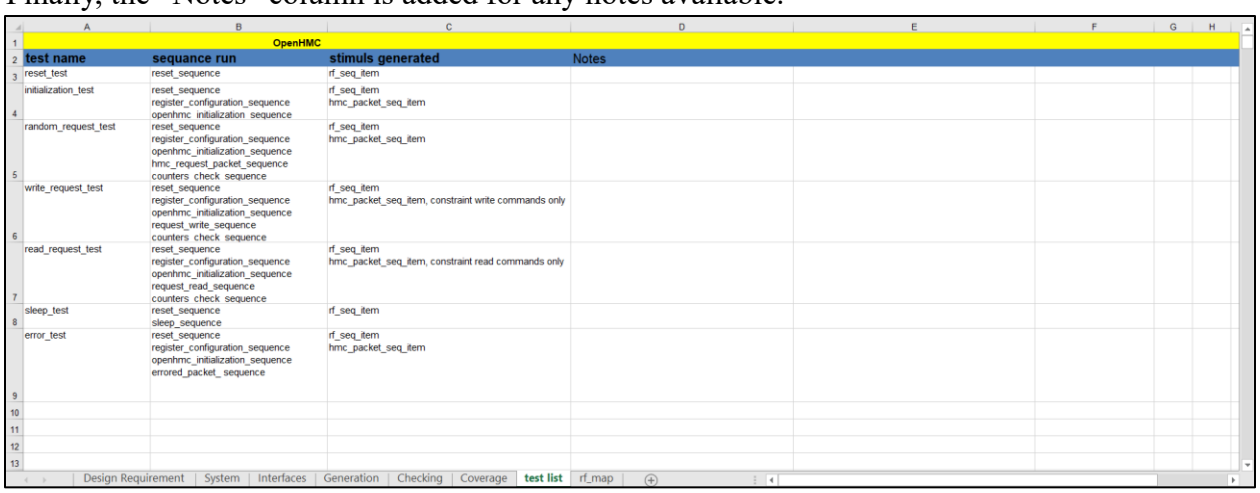
ID	Design Requirement Description		covergroup	coverpoint	cross
	OpenHMC				
	This command is issued by the link master to return tokens when there is no other link				
246	traffic flowing at the time the token is to be returned.				
247	CMD field = 6'b000_010				
248	LNG field = 1				
249	If using response open loop mode, the value in the RTC field is "Don't Care."				
250	**# 4 - INIT RETRY (IRTRY)**				
251	CMD field = 6'b000_011				
252	LNG field = 1				
253	SEQ field should be set to 0				
254	RTC field should be set to 0				
255	Each Start_retry IRTRY packet must have the "StartRetryFlag" set.	FRP[0]=1			
	When the irtry_received_threshold at the Receiver (RX)-Link is reached, the Transmit (TX)	FRP[1]=1			
256	link starts to transmit a series of clear_error IRTRY packets that have the "ClearErrorAbortFlag" set.				
	Each Start_retry IRTRY packet must have ERRSTAT field in its tail with value indicating there is link	ERRSTAT[6:2]= 5'b1000			
257	error.				
	• The link master should always embed the latest FRP it received into the RRP field of				
	each IRTRY packet as it does for any outgoing packet.				
258	• None of the IRTRY packets should be loaded into the Retry Buffer.				
	HMC_RX receives a number of clear_error IRTRY packets equal to or more than				
259	irtry_received_threshold in the register file (default).				
	The TX link sends another series of start_retry packets if the error_abort_mode was not cleared after				
260	256cycles.				
	In case the host exceeds the maximum allowable delay, the HMC retry buffer may run full and				
261	therefore throttle packet streaming which leads to NULL FLIT's between transaction packets.				
	HMC_TX sends a number of Start_retry IRTRY packets equal to or more than irtry_to_send in the				
262	register file (default).				
263	**# notes on some errors**				
	An error is detected whenever there 's a mismatch between CRC field in the packet tail and the CRC	Packet CRC != RX CRC	errored_packetz_cov	1-errored_crc_coverpoint.	4-cross_errored_crc_LNG.
264	calculated at RX.			2-errored_LNG_coverpoint.	5-cross_errored_crc_seq_no.
	An additional source of error is when there is a mismatch between sequence number included in	current seq no != previous seq no +1		3-errored_seq_no_coverpoint.	6-cross_errored_LNG_seq_no.
	each packet and the sequence				
265	number calculated at the RX.				
266	If packet length is more than 17 FLITS or packet length is not correct, an error occurs.	LNG>17 || LNG calculated != LNG received			
267	**# Power on and Initialization**				
	1. connect cli_hmc, and hold hmc_res_n low (active)	since we don't use an actial hmc device, the hmc configuration			
	2. set hmc_res_n = 1 (deactivate)	steps might be skipped. The main focus is on the FSM and the			
	3. load register configuration on HMC using I2C or JTAG	steps when the openHMC performs the initialization			
	4. set hmc_init_cont_set when register load is finished	we should monitor the Status Init register of OpenHMC			
268	5. openHMC performs initialization (NULL1 -> TS1 -> NULL2 -> IDLE)				
	1.Reset/Clear initialization: P_RST_N must be LOW for at least t_RST. LxRXPS for each link must	t_RST = 20 ns (min)			
	be set (HIGH if active, LOW if disabled) for t_IS prior to the de-assertion of P_RST_N	t_IS = 10 ns (min)			
269		Init status of TX link = INIT_TX_NULL1			
	2. Loading HMC device configuration: after loading configuration of HMC, the INIT CONTINUE	this step can be skipped and only set the hmc_init_cont_set register			
	command is issued also at which-the requester must transmit NULL FLITs. in addition, the responder	inside the OpenHMC controller			
270	enters its start state and transmits PRPS stream				
	3. requester keeps sending NULL FLITs to responder and within t_RESP1 scramblers will sync and	t_RESP1 = 1ms (max).			

Design Requirement | System | Interfaces | Generation | Checking | Coverage | test list | rf_map

Figure 4.6: Coverage Sheet

4.7 Test List Sheet

This sheet has 4 columns which are “test name”, “sequence run”, “stimulus generated” and “Notes”. The “test name” column specifies the tests that would be run in the testbench. The “sequence run” column specifies the sequences that would be generated within each test. The “stimulus generated” column specifies the sequence items that would be used in these sequences. Finally, the “Notes” column is added for any notes available.



	test name	sequance run	stimuls generated	Notes
		OpenHMC		
3	reset_test	reset_sequence	rf_seq_item	
4	initialization_test	reset_sequence register_configuration_sequence openhmc_initialization_sequence	rf_seq_item hmc_packet_seq_item	
5	random_request_test	reset_sequence register_configuration_sequence openhmc_initialization_sequence hmc_request_packet_sequence counters_check_sequence	rf_seq_item hmc_packet_seq_item	
6	write_request_test	reset_sequence register_configuration_sequence openhmc_initialization_sequence request_write_sequence counters_check_sequence	rf_seq_item hmc_packet_seq_item, constraint write commands only	
7	read_request_test	reset_sequence register_configuration_sequence openhmc_initialization_sequence request_read_sequence counters_check_sequence	rf_seq_item hmc_packet_seq_item, constraint read commands only	
8	sleep_test	reset_sequence sleep_sequence	rf_seq_item	
9	error_test	reset_sequence register_configuration_sequence openhmc_initialization_sequence errored_packet_sequence	rf_seq_item hmc_packet_seq_item	
10				
11				
12				
13				

Design Requirement | System | Interfaces | Generation | Checking | Coverage | test list | rf_map

Figure 4.7: test list Sheet

4.8 RF Map Sheet

This sheet contains the register file address map and its contents.

	A	B				C	D	E
1		OpenHMC						
2		Design Requirement Description						
3		# Register File Address Map # Register	# Address			# Register Description		
4		status_general	0x0			General HMC Controller Status		
5		status_init	0x1			Debug register for initialization		
6		control	0x2			Control register		
7		sent_p	0x3			Number of posted requests issued		
8		sent_np	0x4			Number of non-posted requests issued		
9		sent_r	0x5			Number of read requests issued		
10		poisoned_packets	0x6			Number of poisoned packets received		
11		rcvd_rsp	0x7			Number of responses received		
12		counter_reset	0x8			Reset all counter		
13		tx_link_retries	0x9			Number of Link retries performed on TX		
14		errors_on_rx	0xA			Number of errors seen on RX		
15		run_length_bit_flip	0xB			Number of bit flips performed due to run length limitation		
16		error_abort_not_cleared	0xC			Number of error_abort_mode not cleared		
17		■ from above table address can be only within 0x0 to 0xC						
18		# Register File Contents - Status General (read only) # Field # Reset	# Bit		# Width [Bits]	# Register Description		
19		link_up	0	1	0	Link is ready for operation		
20		link_training	1	1	0	Link training in progress		
21		sleep_mode	2	1	0	HMC is in Sleep Mode		
22		FERR_N	3	1	0	HMC FERR_N signal		
23		lanes_reversed	4	1	0	0: Normal Operation 1: Lanes reversed (lane 15/8 with 0, ...)		
24		phy_tx_ready	8	1	0	SerDes TX reset is done		
25		phy_rx_ready	8	1	0	SerDes RX reset is done		
26		hmc_tokens_remaining	16+:LOG_MAX_HMC_TOKENS	LOG_MAX_HMC_TOKENS	0	Number of Tokens remaining in the HMC input buffer		
27		rx_tokens_remaining	32+:LOG_MAX_RX_TOKENS	LOG_MAX_RX_TOKENS	0	Number of Tokens remaining in the rx_link input buffer (if param DBG_RX_TOKEN_MON = 1)		
28		lane_polarity_reversed	48+:NUM_LANES	NUM_LANES	0	0: Normal Operation 1: Data is logically inverted lane-by-lane		
29		# Register File Contents - Status Init (read only) # Field # Reset	# Bit		# Width [Bits]	# Register Description		
30		lane_descramblers_locked	0+:NUM_LANES	NUM_LANES	0	Lane by lane descrambler locked		
31		descramblers_part_aligned	16+:NUM_LANES	NUM_LANES	0	Lane by lane descrambler partially aligned		
32		descrambler_aligned	32+:NUM_LANES	NUM_LANES	0	Lane by lane descrambler fully aligned		
33		all_descramblers_aligned	48	1	0	All descramblers are aligned		
		status_init_rx_init_state	51:49	3	0	Init status of the RX link 3'b000: HMC_DOWN 3'b001: HMC_WAIT_FOR_NULL 3'b010: HMC_NULL 3'b011: HMC_TS1_PART_ALIGN 3'b100: HMC_TS1_FIND_REF 3'b101: HMC_TS1_ALIGN		

Design Requirement | System | Interfaces | Generation | Checking | Coverage | test list | rf_map

Figure 4.8: rf_map Sheet

Chapter 5
Testbench Architecture

An imagination of how the architecture will look like is shown in (Figure 5.1).

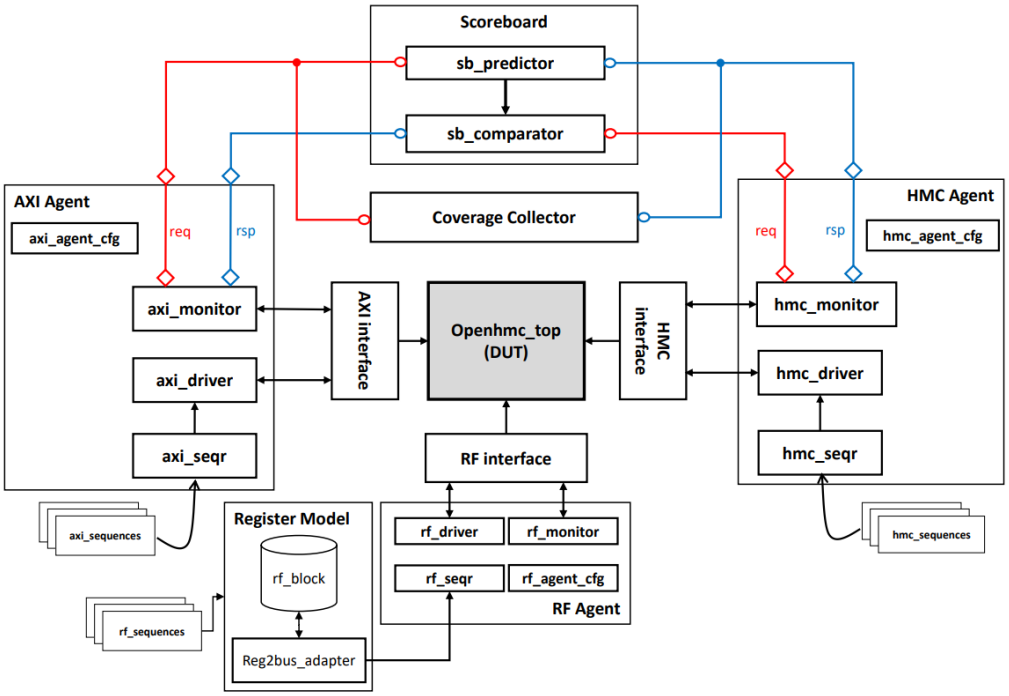


Figure 5.1: Testbench Architecture

5.1 Main Code Files

In this section, some of the main code files will be discussed along with their functionality and role.

5.1.1 tb_top module

This is the top module of the testbench where it contains all the configuration parameters used in the testbench. Also, it contains the instantiations of all the interfaces connected to the DUT (Design Under Test) with the instantiation of the design top module. It is bounded to the assertions file. It imports all the testbench packages. It generates the clock of both hmc agent and axi agent and it performs reset at the start of the test.

5.1.2 Packages

Each package imports the packages under it in the testbench hierarchy and includes a group of related classes in it. However, “cmd_pkg” includes only “the hmc_pkt_item” while it is mainly used to define the commands in this item as string enums.

5.1.3 Environment

It’s the wrapper of all the uvm components in the testbench. It has a connect phase to connect all these components with each other.

5.1.4 Configuration Class

Each configuration class has the configuration of the agent or the environment related to it. The environment configuration class contains the configuration classes of each agent in this environment. The agent configuration classes contain the interface of this agent and determines whether the agent is active (has a driver) or passive (has no driver).

5.1.5 System Interface

This interface provides the system clock and reset to the other interfaces in the environment.

5.1.6 hmc_pkt_item class

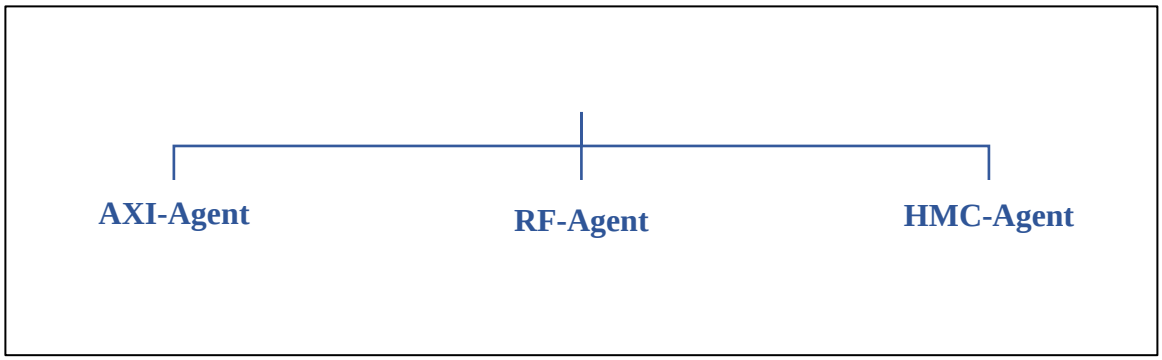
This class is the sequence item representing the packet to be sent and received to and from the dut. This sequence item is generated through the sequences in any agent to represent whether a request sent from the dut or a response sent from the agent starting from the sequences then to the sequencer which delivers it to the agent driver which sends it to the agent interface which passes it to the dut. This item has many fields representing the packet fields and these fields are of random type to be easy to randomize them in the sequence generated to send packets from the AXI agent. The item also has functions to pack or unpack arrays sent to it into packets by filling the fields with the corresponding elements found in the array sent to it, this happens when doing unpacking, and by converting the fields in the packet into an array, this happens when doing packing. The item also has a function to calculate the crc to be sent with the packet and it ’s also used to compare the calculated crc with the crc sent with the packet coming from the dut. Finally, the item has constraints regarding each field in it so that when randomizing this fields in the sequences, they vary randomly within these constraints to manage the randomization to have the values we want in each field.

5.1.7 Other Interfaces

The other interfaces to the dut has an agent for each one of them, and we will talk about these agents in details in the following chapters, these agents are:

Each Agent of them has its own interface file which is responsible for reacting with the dut at the pin level and interact with the driver and monitors of its agent, so we have 3 interface files other than the system interface which are:

- axi_if.
- rf_if.
- hmc_if.



Chapter 6
RF Agent

6.1 Register File Module

The OpenHMC DUT contains a module responsible for handling register files, it is connected to both the RX and TX links of the design. It features three main types of registers: Control, Status, and Counter. Control registers directly affect openHMC's or HMC operation. Status registers can be used to monitor the openHMC status, especially during initialization. Counters allow performance measurement. [4]

While the specs don't exactly mention a register file to be used, it's a necessary addition from the DUT to be able to add some of the features mentioned in the specs and configure them depending on the need like initialization, scrambling and descrambling, lane run length limiting, warm reset and power state management. [5]

6.2 Register File Interface

The Register File module has its own interface on the top module, which read or write operations can be made to access the registers inside. The pins of the interface are shown in (Figure 6.1). Each register has a fixed width of 64bits in total of 13 registers some of them are read and write and some are read-only or write-only. Some contains many reserved bits that are fixed to zero to allow adding more options in the future. [4]

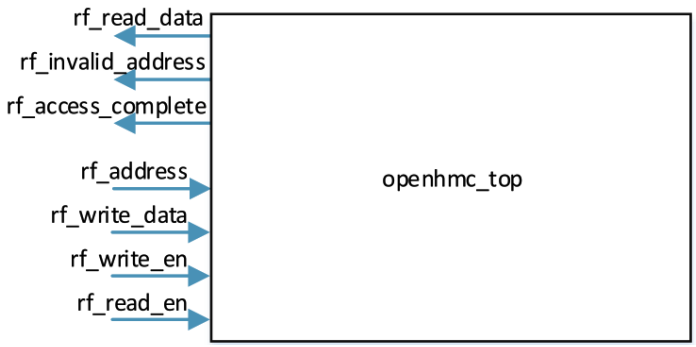


Figure 6.1: register file interface diagram

The driving of the interface is simple but with some constrains. A general read and write operation is shown in (Figure 6.2).

First step is assigning the four-bit address pin to the address of the required register to perform the operation, the address map of the registers is shown in (Table 6.1). Then, either assert write_en or read_en pins but they must not be asserted together. Then we must wait for the assertion of access_complete pin to know that we completed the operation and can sample the output from

read_data if it's a read operation or just know that we finished writing the value inside write_data and move on to next operations.

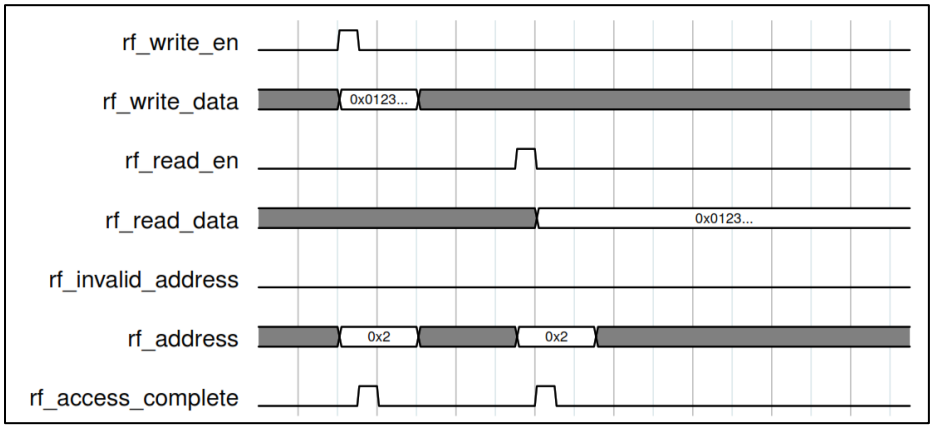


Figure 6.2: register file access, read and write operation

For the invalid address pin, it is asserted in case of wrong address or for example, performing for write operation on a register address that is read-only. [4]

Table 6.1: Register File Address Map

Register	Address	Description
status_general	0x0	General HMC Controller Status
status_init	0x1	Debug register for initialization
control	0x2	Control register
sent_p	0x3	Number of posted requests issued
sent_np	0x4	Number of non-posted requests issued
sent_r	0x5	Number of read requests issued
poisoned_packets	0x6	Number of poisoned packets received
rcvd_rsp	0x7	Number of responses received
counter_reset	0x8	Reset all counter
tx_link_retries	0x9	Number of Link retries performed on TX
errors_on_rx	0xA	Number of errors seen on RX
run_length_bit_flip	0xB	Number of bit flips performed due to run length limitation
error_abort_not_cleared	0xC	Number of error_abort_mode not cleared

6.3 Bad Register File Contents Documenting

The register file contents are listed in appendix B in the OpenHMC documentations. It's in the form of long tables for the three main status and control registers listing the name, size, position, function and accessibility of each field inside these registers. [4]

During testing the registers, 2 of these register's documenting were found to be wrong or outdated that it doesn't match the RTL codes.

First is the Control register, the design documentation of it is shown in (Table 6.2). What is found in the RTL that some of these fields didn't exist in it and an additional field isn't listed here. A snapshot of the code of register file module for the control register is shown in (Figure 6.3). You can now notice the difference that 3 of the fields mentioned in the OpenHMC documentation are missing with are: debug_dont_send_tret, debug_halt_on_error_abort and debug_halt_on_tx_retry. And for the RTL a field named warm_reset isn't mentioned in the documentation.

Table 6.2: Control register documentation of OpenHMC

Field	Bit	Width [Bits]	Description & Encoding	Res	HW	SW
p_rst_n	0	1	Active low HMC reset	0	ro	rw
hmc_init_cont_set	1	1	Allow descramblers to lock	0	ro	rw
set_hmc_sleep	2	1	Request HMC sleep mode. Sleep mode can be monitored by the 'sleep_mode' field in the Status General Register	0	ro	rw
scrambler _disable	3	1	Disable Scrambler and Descrambler for testing purposes	0	ro	rw
run_length _enable	4	1	Disable the run length limiter in the TX scrambler logic	0	ro	rw
first _cube_ID	7:5	3	Set the Cube ID of the first HMC connected. Used in irtry packets	0	ro	rw
debug _dont_send_tret	8	1	Prohibit controller from sending any TRET packets	0	ro	rw
debug _halt_on _error_abort	9	1	HALT tx_link after rx_link entered error abort	0	ro	rw
debug _halt_on _tx_retry	10	1	HALT tx_link after performing a retry	0	ro	rw
rx_token _count	16+: LOG_MAX_RX_TOKENS	10	Set the input buffer space in the RX block	{LOG_MAX_RX_TOKENS{1'b1}}	ro	rw
irtry_received _threshold	36:32	5	Set the number of irtry packets to be received to trigger retry	0x10	ro	rw
irtry_to _send	44:40	5	Set the number of irtry to be sent	0x18	ro	rw

```
271     //Register: control
272     `ifdef ASYNC_RES
273     always @(posedge clk or negedge res_n) `else
274     always @(posedge clk) `endif
275     begin
276
277         if(!res_n)
278         begin
279             control_p_rst_n              <= 1'h0;
280             control_hmc_init_cont_set    <= 1'b0;
281             control_set_hmc_sleep        <= 1'h0;
282             control_warm_reset           <= 1'h0;
283             control_scrambler_disable    <= 1'h0;
284             control_run_length_enable    <= 1'h0;
285             control_rx_token_count       <= {LOG_MAX_RX_TOKENS{1'b1}};
286             control_irtry_received_threshold    <= 5'h10;
287             control_irtry_to_send        <= 5'h18;
288         end
289         else
```

Figure 6.3: Control register RTL code

The second wrong documentation for a register is for the counter_reset register, it consists of 1 bit that we should flag to this register to tell it that we want to reset all the openhmc counters. In the documentation as shown in (Table 6.3) it is mentioned as read-only. Meanwhile, in the rtl as shown in (Figure 6.5) and (Figure 6.4) its function is triggered during write operation and the invalid address pin is asserted if a read operation is performed.

Table 6.3: counter_reset documentation in openhmc

Field	# Bits	Description & Encoding	Reset	HW	SW
tx_link_retries	32	Incremental 1-bit counter: Number of Link retries performed on TX	0	wo	ro
errors_on_rx	32	Incremental 1-bit counter: Number of successful HMC retries performed	0	wo	ro
run_length _bit_flip	32	Incremental 1-bit counter: How many bit_flips were performed by the run length limiter	0	wo	ro
error_abort _not_cleared	32	Incremental 1-bit counter: Indicates the number of link retry attemps that timed out	0	wo	ro
counter_reset	1	Reset counter in the 'Other Counter' category. This bit is automatically cleared	0	wo	ro

```
550                 4'h8:
551                 begin
552                     read_data       <= {HMC_RF_RWIDTH{1'b0}};
553                     invalid_address <= read_en;
554                     access_complete <= read_en || write_en;
555                 end
```

Figure 6.4: invalid address pin behavior when reading from counter reset register

```
431     //Register: counter_reset
432     `ifdef ASYNC_RES
433     always @(posedge clk or negedge res_n) `else
434     always @(posedge clk) `endif
435     begin
436
437         if(!res_n)
438         begin
439             rreinit <= 1'b0;
440         end
441         else
442         begin
443
444             if((address == 8) && write_en)
445             begin
446                 rreinit <= 1'b1;
447             end
448             else
449             begin
450                 rreinit <= 1'b0;
451             end
452         end
453
454     end
```

Figure 6.5: Counter reset register RTL code

6.4 RF Agent Implementation

Verifying register is common in UVM and has a systematic flow. A known term called "Register Abstraction Layer" it is explained in the UVM cookbook and it is the method used to write the codes of the register file agent and to build a register block inside the testbench that maps all the registers inside the DUT with all their fields. A general diagram for the register model and agent is shown in (Figure 6.6). [6]

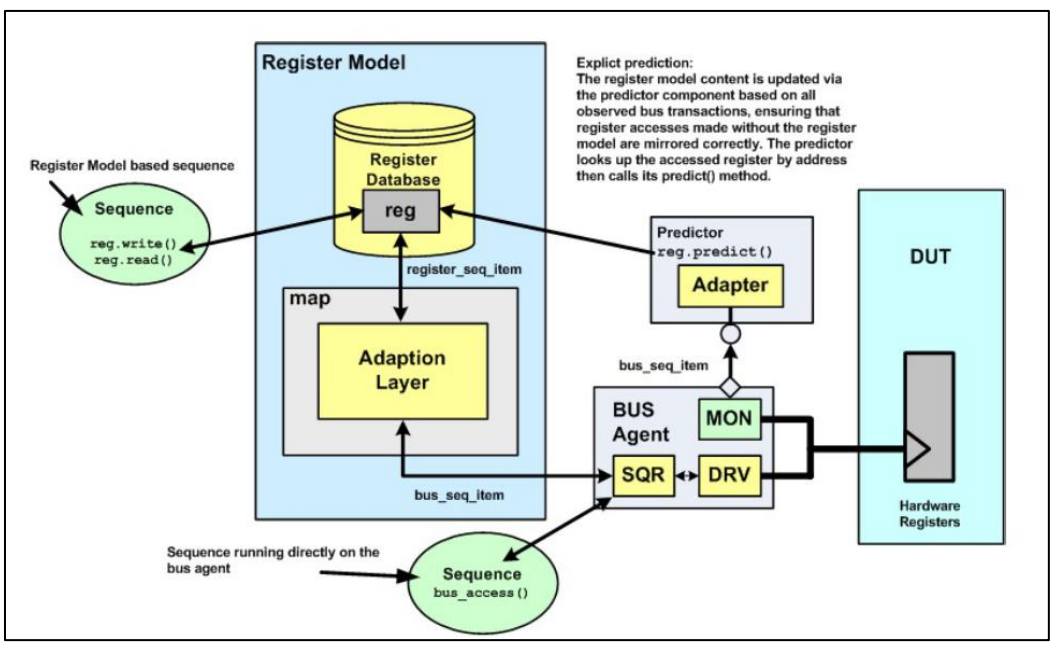


Figure 6.6: diagram of uvm register model and agent

The steps followed is first write a separate class for each register, write a register block with a handle to each of these register's classes, inside the block implement a map and list each register in a specific position and assign it to an address. Then, an adaptor is needed to be the bridge between the register layer and the actual agent that communicates with the interface of the DUT. Below is brief explanation of these steps.

6.4.1 Registers

UVM has already built-in classes to implement registers and their fields, all we have to do is extend from them and call a certain function to configure each register and map the fields into their positions inside the register. The base classes are shown in (Figure 6.7).

uvm_reg

Register abstraction base class

CLASS HIERARCHY

uvm_void
uvm_object
uvm_reg

CLASS DECLARATION

virtual class uvm_reg extends uvm_object

INITIALIZATION

new	Create a new instance and type-specific configuration
configure	Instance-specific configuration
set_offset	Modify the offset of the register

uvm_reg_field

Field abstraction class

CLASS HIERARCHY

uvm_void
uvm_object
uvm_reg_field

CLASS DECLARATION

class uvm_reg_field extends uvm_object

value	Mirrored field value.

INITIALIZATION

new	Create a new field instance
configure	Instance-specific configuration

Figure 6.7: UVM base classes for registers

For example, a snapshot of how the code of one of the registers looks like in (Figure 6.8)

```
class reg_status_init extends  uvm_reg;

   `uvm_object_utils(reg_status_init)

   rand uvm_reg_field  lane_descramblers_locked;
   rand uvm_reg_field  descrambler_part_alligned;
   rand uvm_reg_field  descrambler_alligned;
   rand uvm_reg_field  all_descramblers_alligned;
   rand uvm_reg_field  status_init_rx_init_state;
   rand uvm_reg_field  status_init_tx_init_state;


function new(string name = "");
   super.new(name,64,UVM_NO_COVERAGE);
endfunction : new

virtual function void build();

   // Create object instance for each field
   this.lane_descramblers_locked  = uvm_reg_field::type_id::create("lane_descramblers_locked");
   this.descrambler_part_alligned = uvm_reg_field::type_id::create("descrambler_part_alligned");
   this.descrambler_alligned      = uvm_reg_field::type_id::create("descrambler_alligned");
   this.all_descramblers_alligned = uvm_reg_field::type_id::create("all_descramblers_alligned");
   this.status_init_rx_init_state = uvm_reg_field::type_id::create("status_init_rx_init_state");
   this.status_init_tx_init_state = uvm_reg_field::type_id::create("status_init_tx_init_state");

   // Configure each field (parent, size, lsb_pos, access, volatile, reset, has_reset, is_rand, individually_accessible)
   this.lane_descramblers_locked.configure  (this, 8,  0, "RO", 0,  8'h0, 1, 0, 1);
   this.descrambler_part_alligned.configure (this, 8, 16, "RO", 0,  8'h0, 1, 0, 1);
   this.descrambler_alligned.configure      (this, 8, 32, "RO", 0,  8'h0, 1, 0, 1);
   this.all_descramblers_alligned.configure (this, 1, 48, "RO", 0,  'h0, 1, 0, 1);
   this.status_init_rx_init_state.configure (this, 3, 49, "RO", 0, 3'h0, 1, 0, 1);
   this.status_init_tx_init_state.configure (this, 2, 52, "RO", 0, 2'h0, 1, 0, 1);

endfunction : build

endclass : reg_status_init
```

Figure 6.8: a snapshot of status init register class

6.4.2 Register Block

Here UVM has a class named uvm_reg_block shown in (Figure 6.9) and doing the same thing for the registers, now the register block will have a handle for each of these registers and map them to an address.

uvm_reg_block

Block abstraction base class

Class Hierarchy

- uvm_void
- uvm_object
- **uvm_reg_block**

Class Declaration

```
virtual class uvm_reg_block extends uvm_object
```

default_path	Default access path for the registers and memories in this block.

Initialization

new	Create a new instance and type-specific configuration
configure	Instance-specific configuration
create_map	Create an address map in this block
check_data_width	Check that the specified data width (in bits) is less than or equal to the value of `UVM_REG_DATA_WIDTH
set_default_map	Defines the default address map
default_map	Default address map
lock_model	Lock a model and build the address map.
is_locked	Return TRUE if the model is locked.

Figure 6.9: snapshot of uvm_reg_block class

6.4.3 Adapter

The adapter is responsible to convert from the register sequence item, into the item used inside the agent to communicate with the interface. The structure of the code is very standard containing only 2 functions named reg2bus and bus2reg each must be written in a certain way to work properly. Adapter base class and the 2 standard functions are shown in (Figure 6.10)

uvm_reg_adapter

This class defines an interface for converting between uvm_reg_bus_op and a specific bus transaction.

Class Hierarchy

- uvm_void
- uvm_object
- **uvm_reg_adapter**

Class Declaration

```
virtual class uvm_reg_adapter extends uvm_object
```

new	Create a new instance of this type, giving it the optional *name*.
supports_byte_enable	Set this bit in extensions of this class if the bus protocol supports byte enables.
provides_responses	Set this bit in extensions of this class if the bus driver provides separate response items.
parent_sequence	Set this member in extensions of this class if the bus driver requires bus items be executed via a particular sequence base type.
reg2bus	Extensions of this class *must* implement this method to convert the specified uvm_reg_bus_op to a corresponding uvm_sequence_item subtype that defines the bus transaction.
bus2reg	Extensions of this class *must* implement this method to copy members of the given bus-specific *bus_item* to corresponding members of the provided *bus_rw* instance.

reg2bus

```
pure virtual function uvm_sequence_item reg2bus(
    const ref uvm_reg_bus_op rw
)
```

Extensions of this class *must* implement this method to convert the specified uvm_reg_bus_op to a corresponding uvm_sequence_item subtype that defines the bus transaction.

The method must allocate a new bus-specific uvm_sequence_item, assign its members from the corresponding members from the given generic *rw* bus operation, then return it.

bus2reg

```
pure virtual function void bus2reg(
        uvm_sequence_item bus_item,
    ref uvm_reg_bus_op    rw
)
```

Extensions of this class *must* implement this method to copy members of the given bus-specific *bus_item* to corresponding members of the provided *bus_rw* instance. Unlike reg2bus, the resulting transaction is not allocated from scratch. This is to accommodate applications where the bus response must be returned in the original request.

Figure 6.10: register adaptor base class

6.5 RF Interface Assertions

The assertions are written to make sure the pins are driven correctly like the way it is explained in the documentations. For the register file, we wrote 4 assertions for all the possible cases we could think of. The names of these assertions, code and explanations are shown in (Table 6.4) and (Figure 6.11) also noting that the type of these assertions are "concurrent" which means that it runs during all the simulation time, it might trigger if the first statement is true or it might check for a statement to be true every clock cycle.

Table 6.4: Register File Assertions Explanation

Assertion Name	Explanation
no_simultaneous_read_and_write	As stated in the design documentation, the read enable and write enable signals must not be asserted at the same time, so the register file accepts only 1 operation at a time to access it. Either read or write
invalid_address_assert_if_wrong_write_operation	Inside of it written all the addresses that its registers are read-only, so if write enable signal is asserted with these registers' addresses the invalid address signal should be asserted
invalid_address_assert_if_wrong_read_operation	Same as above but for the 1 register that is write only
access_complete_one_cycle_after_operation	By testing and as shown in the design documentation, access complete signal is always asserted after 1 clock cycle of asserting read or write enable

```
141 //-----------------------------------------------------------------------------------------//
142 //---------------------------- Register File Assertions ------------------------------------//
143 //-----------------------------------------------------------------------------------------//
144 no_simultaneous_read_and_write :
145   `assert_clk ( ((rf_write_en && rf_read_en) == 0) )
146
147 invalid_address_assert_if_wrong_write_operation :
148   `assert_clk ( (rf_write_en && (rf_address==4'h0 || rf_address==4'h1 || rf_address==4'h3 || rf_address==4'h4
149                  || rf_address==4'h5  || rf_address==4'h6  || rf_address==4'h7  || rf_address==4'h9  || rf_address==4'hA
150                  || rf_address==4'hB  || rf_address==4'hC) ) |=> rf_invalid_address )
151
152 invalid_address_assert_if_wrong_read_operation :
153   `assert_clk ( (rf_read_en && rf_address==4'h8) |=> rf_invalid_address )
154
155 access_complete_one_cycle_after_operation :
156   `assert_clk ( (rf_write_en || rf_read_en) |=> rf_access_complete )
157
```

Figure 6.11: Register File Assertions codes

The codes of the assertions are written in the macro style. It is basically like writing a function and passing an argument to it to perform the assertion operation. It is explained more in [7]

6.6 RF Sequences

The register file is a very major component in the DUT, it depends on it to do so many operations and to do transitions of its finite state machines depending on the signals driven on its control register, and we can monitor the status and finite state machine of the DUT from the register file also. So, the register file sequences are a main role player in all of our tests to know the proper timing of doing some operations. So far, the written sequences codes are all based on the current tests so if more tests are developed that needs more sequences other than the current ones, the sequences will increase to achieve the needed scenarios. In addition, we note that all the sequences are register model based, as when a read or write operation is required, we write the functions that operate on the register block then the adapter is responsible to deliver it to the agent and not create an item then pass it to the driver directly. In addition, all the read/write operations are "front-door" that to perform the operation it uses the normal way to access the interface and drive the stimulus if it's a backdoor operation it will access the registers from the DUT directly skipping the interface which is not implemented here. In this section, the main sequences are explained.

6.6.1 Initialization Sequence

The most important register file sequence, must be run at the start of every test and without it the DUT won't work probably. It operates right after any reset in the system or in the very beginning after the first reset. To explain it. It can mainly be divided into four steps. First is to configure the DUT for example set the input buffer space in the RX block, configure it if we are going to disable or enable the scramblers and two more parameters related to irtry operation. As shown in (Figure 6.12) the first write operation in the control register.

```
# ******************************
#   CONTROL REGISTER
# ******************************
#    p_rst_n=0,
#    hmc_init_cont_set=0,
#    set_hmc_sleep=0,
#    warm_reset=0,
#    scrambler_disable=1,
#    run_length_enable=1,
#    rx_token_count=75,
#    irtry_received_threshold=10,
#    irtry_to_send=18
# ************************************************************
```

Figure 6.12: first step in initialization sequence

The three numbers at the end are extracted from the hmc agent configuration object. They are either manually set or in other approach which is not done in our tests is to randomize the values inside the configuration object. Next step is to check if the DUT asserted the ready signal, we can

monitor it inside the status general register so in the code we wrote a loop to keep reading from this register till the ready signal is asserted as shown in (Figure 6.13).

```
//-- Wait until the PHY is out of reset
while (phy_tx_ready == 1'b0) begin
  #1us;
  rf_rb.m_reg_status_general.read(status, data, .parent(this));
  phy_tx_ready = rf_rb.m_reg_status_general.phy_tx_ready.get();
  `uvm_info("INITIALiZATION_SEQ", "Waiting for the PHY TX to get ready", UVM_NONE)
end
`uvm_info("INITIALiZATION_SEQ", "Phy TX ready", UVM_NONE)
```

Figure 6.13: waiting for phy_tx_ready in initialization sequence

Next step is to make the hmc agent out of reset mode, because it must wait for the P_RST_N signal to be asserted which is different from the reset of the system, this one is driven by the DUT itself as input to the hmc agent as stated in the specs [5]. After asserting it, the hmc initialize internally and then assert the physical ready signal which can be monitored by the status general register. Checked by a while loop as well and shown in (Figure 6.14) the contents of status general register after asserting P_RST_N and hmc asserts its ready signal.

```
# ********************************
#   STATUS GENERAL REGISTER
# ********************************
#    link_up=0,
#    link_training=1,
#    sleep_mode=0,
#    FERR_N=1,
#    lanes_reversed=0,
#    phy_tx_ready=1,
#    phy_rx_ready=1,
#    hmc_tokens_remaining=0,
#    rx_tokens_remaining=75,
#    lane_polarity_reversed=0
# ****************************************************************
```

Figure 6.14: status register field values during initialization and training

Now all is left is waiting for both DUT and hmc agent to go through all initialization steps which are NULL_1 then TS1 then NULL2 then IDLE, after it reaches idle the link_up signal inside the status general register is asserted. The hmc as well after reaching the idle state will enter the initial TRET state that sends the tokens inside its buffer, in our implementation we didn't actually implement a buffer so we send random token count. But before that, just after idle state the status general contents are shown in (Figure 6.15).

```
# ******************************
#   STATUS GENERAL REGISTER
# ******************************
#    link_up=1,
#    link_training=0,
#    sleep_mode=0,
#    FERR_N=1,
#    lanes_reversed=0,
#    phy_tx_ready=1,
#    phy_rx_ready=1,
#    hmc_tokens_remaining=0,
#    rx_tokens_remaining=75,
#    lane_polarity_reversed=0
# ****************************************************************
```

Figure 6.15: status general content after link is up

Another approach could be taken here, instead of monitoring the status general register. For more in detail debugging we could monitor the status init register which contains the information about 2 state machines for the DUT's tx link and its rx link that see what the hmc agent sends to it and reflects what is see on the fsm. But it is not implemented here. On the other hand, the hmc monitor will be responsible to report the state of both DUT and hmc agent during this initialization, shown in (Figure 6.16) how the monitor report the states during simulation. And with that, the initialization sequence is complete.

```
@ 3536001: uvm_test_top.m_env.m_rf_agent.m_seqr@@init [INITIALiZATION_SEQ] Waiting for the PHY TX to get ready
@ 4540801: uvm_test_top.m_env.m_rf_agent.m_seqr@@init [INITIALiZATION_SEQ] Waiting for the PHY TX to get ready
@ 4540801: uvm_test_top.m_env.m_rf_agent.m_seqr@@init [INITIALiZATION_SEQ] Phy TX ready
1) @ 4547200: uvm_test_top.m_env.hmc_agent_h.m_req_monitor [HMC_AGENT_MONITOR_link_states()] Requester Link current State: PRBS   vector: 110000
1) @ 4547200: uvm_test_top.m_env.hmc_agent_h.m_rsp_monitor [HMC_AGENT_MONITOR_link_states()] Responder Link current State: POWER_DOWN     vector: 100000
1) @ 4548650: uvm_test_top.m_env.hmc_agent_h.m_req_monitor [HMC_AGENT_MONITOR_link_states()] Requester Link current State: NULL_FLITS     vector: 111000
1) @ 5552000: uvm_test_top.m_env.hmc_agent_h.m_rsp_monitor [HMC_AGENT_MONITOR_link_states()] Responder Link current State: PRBS   vector: 110000
1) @ 5784450: uvm_test_top.m_env.hmc_agent_h.m_rsp_monitor [HMC_AGENT_MONITOR_link_states()] Responder Link current State: NULL_FLITS     vector: 111000
1) @ 6102450: uvm_test_top.m_env.hmc_agent_h.m_req_monitor [HMC_AGENT_MONITOR_link_states()] Requester Link current State: TS1    vector: 111100
0) @ 6108850: uvm_test_top.m_env.hmc_agent_h.m_req_monitor [HMC_AGENT_MONITOR_collect_flits()] Requester Link is not reversed
@ 6553601: uvm_test_top.m_env.m_rf_agent.m_seqr@@init [INITIALiZATION_SEQ] Waiting for PHY RX to get ready
1) @ 6814950: uvm_test_top.m_env.hmc_agent_h.m_rsp_monitor [HMC_AGENT_MONITOR_link_states()] Responder Link current State: TS1    vector: 111100
0) @ 6821350: uvm_test_top.m_env.hmc_agent_h.m_rsp_monitor [HMC_AGENT_MONITOR_collect_flits()] Responder Link is not reversed
@ 7558401: uvm_test_top.m_env.m_rf_agent.m_seqr@@init [INITIALiZATION_SEQ] Waiting for PHY RX to get ready
) @ 7558401: uvm_test_top.m_env.m_rf_agent.m_seqr@@init [INITIALiZATION_SEQ]

) @ 7558401: uvm_test_top.m_env.m_rf_agent.m_seqr@@init [INITIALiZATION_SEQ] Phy RX is ready
1) @ 7700850: uvm_test_top.m_env.hmc_agent_h.m_req_monitor [HMC_AGENT_MONITOR_link_states()] Requester Link current State: NULL_FLITS_2   vector: 111110
1) @ 7709350: uvm_test_top.m_env.hmc_agent_h.m_rsp_monitor [HMC_AGENT_MONITOR_link_states()] Responder Link current State: NULL_FLITS_2   vector: 111110
1) @ 7724850: uvm_test_top.m_env.hmc_agent_h.m_req_monitor [HMC_AGENT_MONITOR_link_states()] Requester Link current State: LINK_UP    vector: 111111
1) @ 7733350: uvm_test_top.m_env.hmc_agent_h.m_rsp_monitor [HMC_AGENT_MONITOR_link_states()] Responder Link current State: LINK_UP    vector: 111111
) @ 7740801: uvm_test_top.m_env.m_rf_agent.m_seqr@@init [INITIALiZATION_SEQ]
```

Figure 6.16: hmc monitor reporting during initialization

6.6.2 Sleep Sequence

This sequence is used to cover 2 states or a certain scenario of the DUT which is also mentioned in the specs, called the sleep or power-down state. [5]

Shown in (Figure 6.17) the specific states and scenario to cover of the DUT's fsm

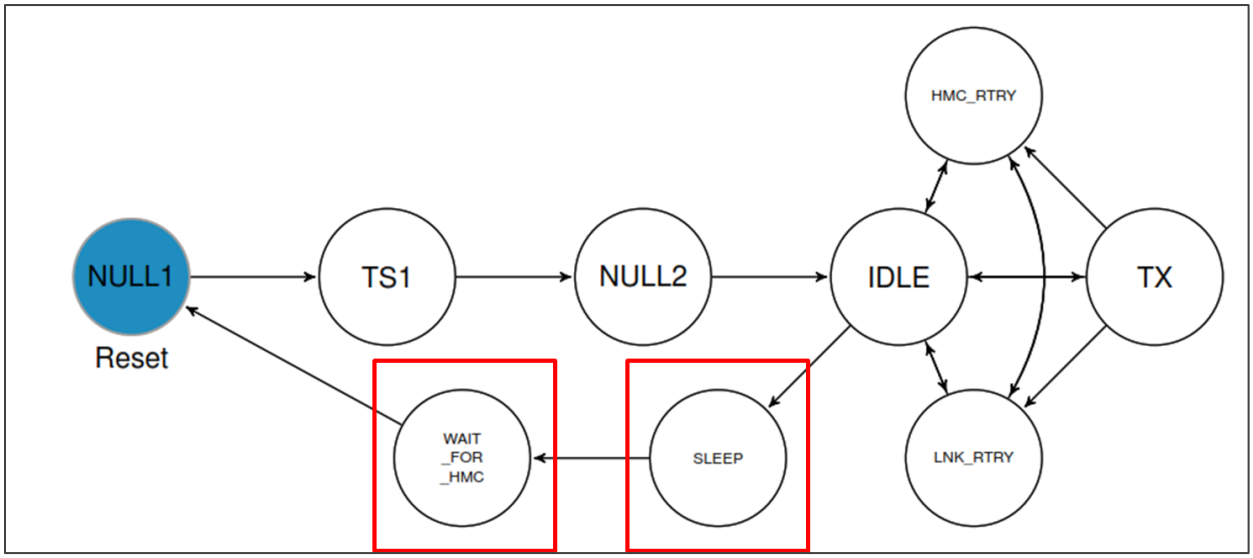


Figure 6.17: sleep scenario inside the DUT fsm

As before to enter this sleep state we must assert a certain field inside the control register called set_hmc_sleep. The content of the control register after this is shown in (Figure 6.18).

```
# ********************************
#   CONTROL REGISTER (before sleep)
# ********************************
#    p_rst_n=1,
#    hmc_init_cont_set=0,
#    set_hmc_sleep=0,
#    warm_reset=0,
#    scrambler_disable=1,
#    run_length_enable=1,
#    rx_token_count=75,
#    irtry_received_threshold=10,
#    irtry_to_send=18
# ****************************************
```

```
# ********************************
#   CONTROL REGISTER (after sleep)
# ********************************
#    p_rst_n=1,
#    hmc_init_cont_set=0,
#    set_hmc_sleep=1,
#    warm_reset=0,
#    scrambler_disable=1,
#    run_length_enable=1,
#    rx_token_count=75,
#    irtry_received_threshold=10,
#    irtry_to_send=18
# ****************************************
```

Figure 6.18: control register in sleep sequence

For the sleep mode, a certain signal from the DUT must be driven to low which is called LxRXPS, as shown in (Figure 6.19) this signal is changed to low right after writing in the control register.

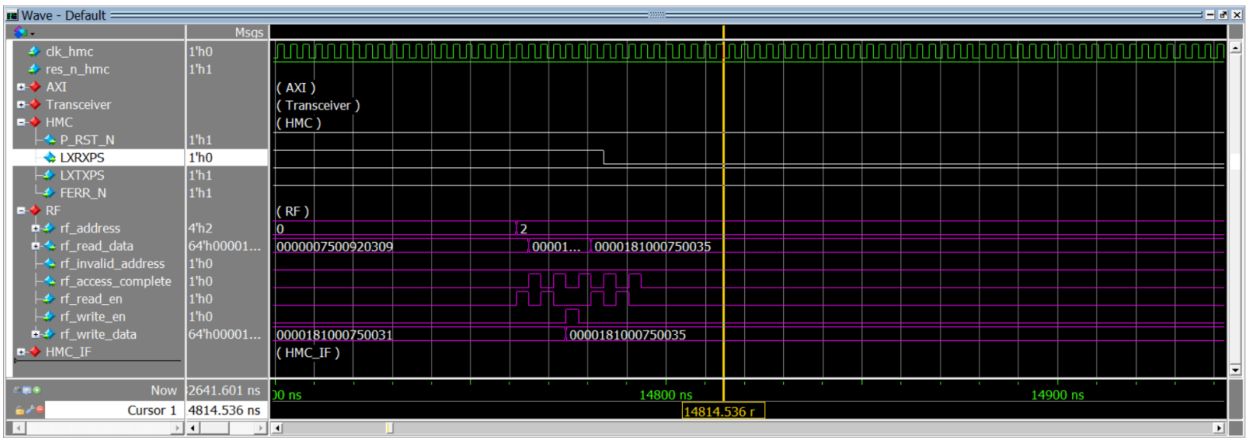


Figure 6.19: LxRXPS is driven to low during sleep

After a while according to the specs [5] the other side must assert the LxTXPS which is driven by the hmc agent, this is shown in (Figure 6.20). After it is driven it sets the DUT into sleep mode and the status register content will be as shown in (Figure 6.21).

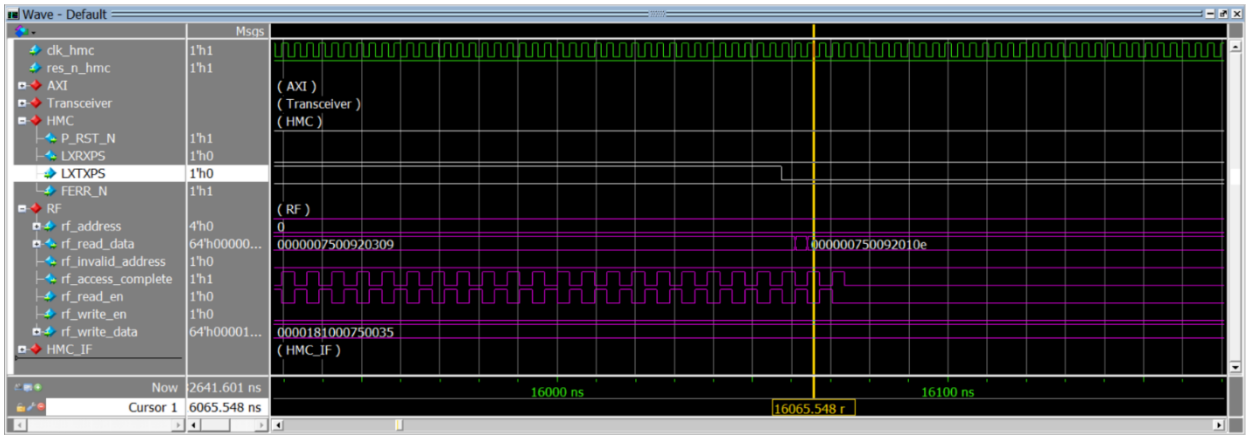


Figure 6.20: the hmc agent drives LxTXPS to low

```
# ********************************
#   STATUS GENERAL REGISTER (during sleep)
# ********************************
#    link_up=0,
#    link_training=1,
#    sleep_mode=1,
#    FERR_N=1,
#    lanes_reversed=0,
#    phy_tx_ready=1,
#    phy_rx_ready=0,
#    hmc_tokens_remaining=92,
#    rx_tokens_remaining=75,
#    lane_polarity_reversed=0
# ******************************************************************
```

Figure 6.21: status register content during sleep

Next step is to exit this sleep mode, just by setting the set_hmc_sleep back to zero in the control register, the DUT asserts back the LxRXPS and enters the Wait for HMC state that it waits for the LxTXPS to be asserted as well as shown in (Figure 6.22).

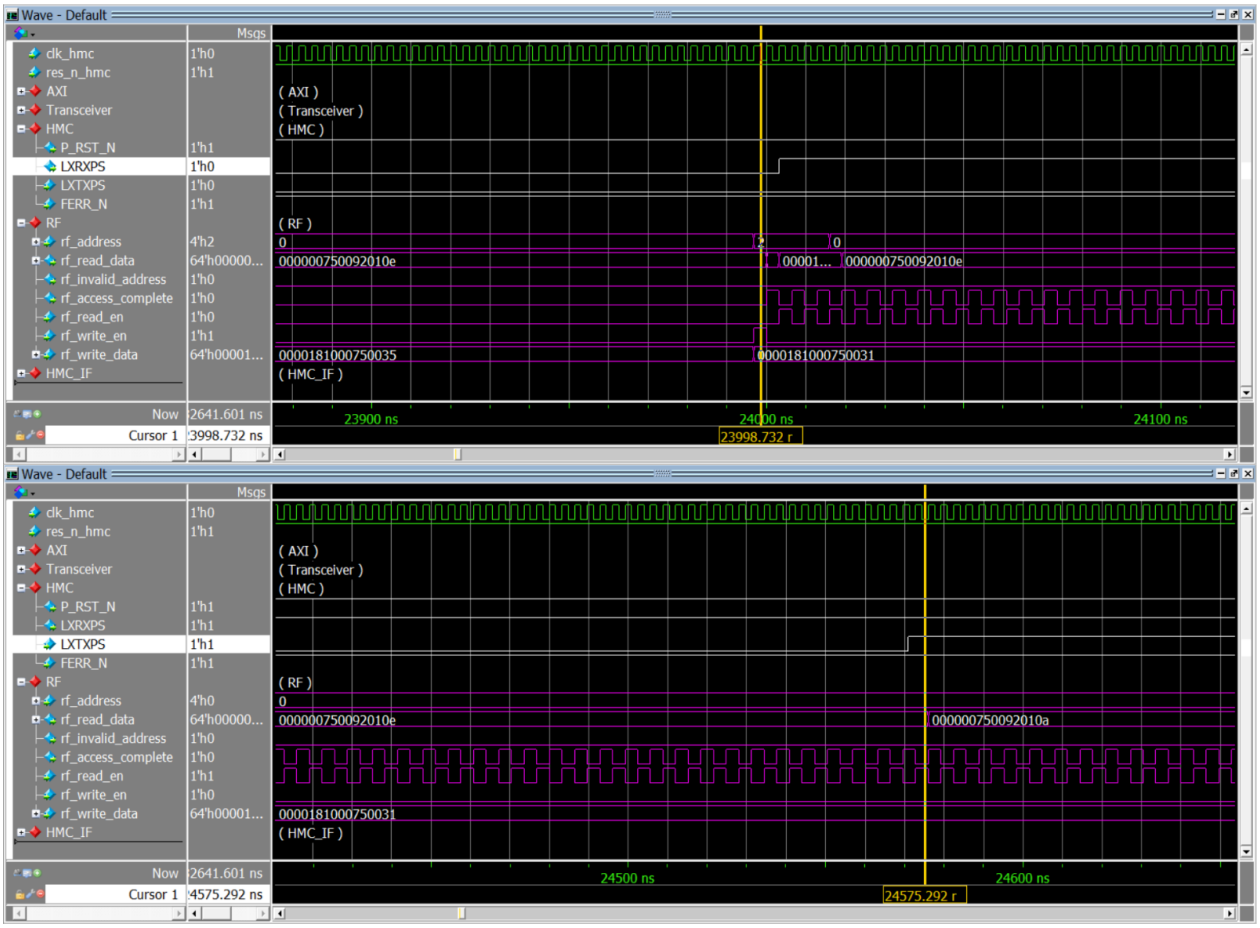


Figure 6.22: exiting sleep mode

Now the agent and DUT go through the initialization again and for the sequence there is nothing left to do but to wait for the link is up field inside the status general to be asserted to continue to the rest of sequences if any is used in the test.

6.6.3 Wrong Operation Sequence

This sequence is specifically written to cover the two assertions mentioned and explained before which are named

- invalid_address_assert_if_wrong_write_operation
- invalid_address_assert_if_wrong_read_operation

Because it is not usual to just casually drive a wrong address for a write or read while performing the normal test, so this sequence is used to do this to intentionally hit the assertion. Since the sequence code is small it can be shown in (Figure 6.23) as well as the simulation waves in (Figure

6.24) to notice the invalid address signal responding to these operations. In addition, the results of the assertions can be shown in (Figure 6.25).

```
class rf_wrong_op_seq extends  base_seq;


  `uvm_object_utils(rf_wrong_op_seq)
  extern function new (string name = "");
  extern task body();

endclass : rf_wrong_op_seq


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

task rf_wrong_op_seq::body();
  super.body();

  rf_rb.m_reg_status_general.write(status, 64'h0, .parent(this));
  rf_rb.m_reg_status_init.write(status, 64'h0, .parent(this));
  rf_rb.m_reg_sent_p.write(status, 64'h0, .parent(this));
  rf_rb.m_reg_sent_np.write(status, 64'h0, .parent(this));
  rf_rb.m_reg_sent_r.write(status, 64'h0, .parent(this));
  rf_rb.m_reg_poisoned_packets.write(status, 64'h0, .parent(this));
  rf_rb.m_reg_rcvd_rsp.write(status, 64'h0, .parent(this));
  rf_rb.m_reg_tx_link_retries.write(status, 64'h0, .parent(this));
  rf_rb.m_reg_errors_on_rx.write(status, 64'h0, .parent(this));
  rf_rb.m_reg_run_length_bit_flip.write(status, 64'h0, .parent(this));
  rf_rb.m_reg_error_abort_not_cleared.write(status, 64'h0, .parent(this));

  rf_rb.m_reg_counter_reset.read(status, data, .parent(this));

endtask : body
```

Figure 6.23: code of the register file wrong op seq

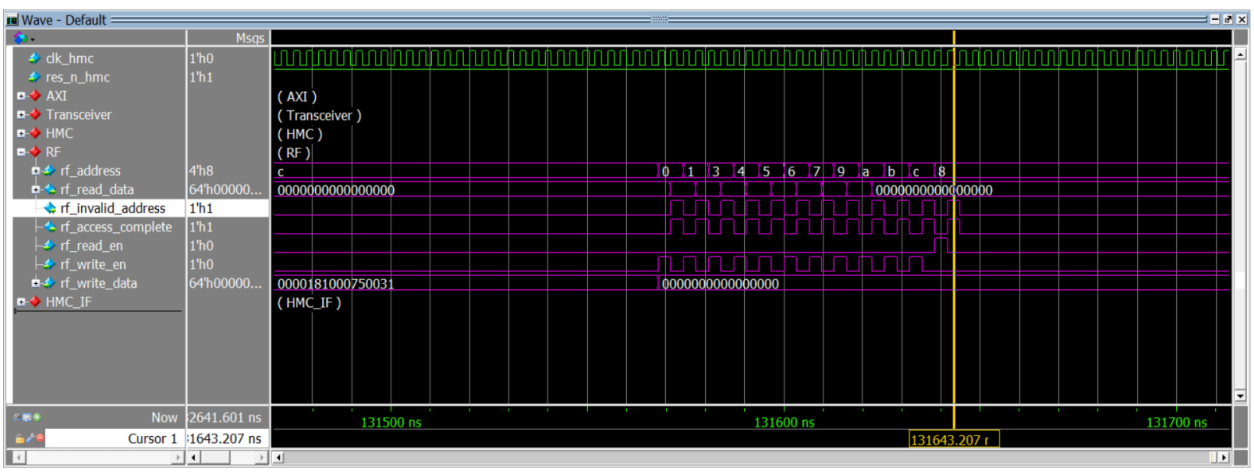


Figure 6.24: the response of the DUT invalid address during performing wrong operations

Assertions

Name	Assertion Type	Language	Enable	Failure Count	Pass Count
/tb_top/dut/openhmc_sva_1/tx_valid_hold_until_ready_active	Concurrent	SVA	on	0	42
/tb_top/dut/openhmc_sva_1/rx_valid_hold_until_ready_active	Concurrent	SVA	on	0	70
/tb_top/dut/openhmc_sva_1/tx_user_hold_until_ready_active	Concurrent	SVA	on	0	42
/tb_top/dut/openhmc_sva_1/rx_user_hold_until_ready_active	Concurrent	SVA	on	0	70
/tb_top/dut/openhmc_sva_1/tx_data_hold_until_ready_active	Concurrent	SVA	on	0	42
/tb_top/dut/openhmc_sva_1/rx_data_hold_until_ready_active	Concurrent	SVA	on	0	70
/tb_top/dut/openhmc_sva_1/no_simultaneous_read_and_write	Concurrent	SVA	on	0	105912
/tb_top/dut/openhmc_sva_1/invalid_address_assert_if_wrong_write_operation	Concurrent	SVA	on	0	11
/tb_top/dut/openhmc_sva_1/invalid_address_assert_if_wrong_read_operation	Concurrent	SVA	on	0	1
/tb_top/dut/openhmc_sva_1/access_complete_one_cycle_after_operation	Concurrent	SVA	on	0	4924
/uvm_pkg/uvm_reg_map/do_write/#ublk#21518159#1731/immed__1735	Immediate	SVA	on	0	0

Figure 6.25: assertions related to rf wrong op seq

6.6.4 Read and Reset Sequences

Some sequences are a simple read to the registers which might be needed to just print it during simulation for debugged. The sequences for the reset were used to cover certain transitions in the fsm of the DUT. Using 2 registers the status init and the status general when we hit a certain state, we enter the reset and that's how we cover it and it is the only way to know exactly the correct timing of the reset not just doing it randomly. For example, to check it we listed the states inside the sequence code and kept reading from the register, once the state is hit, we enter reset. An example is shown in (Figure 6.26). Then continue to initialize and check for another state and reset again and so on.

```
class reset4_seq extends  base_seq;

  typedef enum bit [2:0] {
    HMC_DOWN          = 3'b000,
    HMC_WAIT_FOR_NULL  = 3'b001, //cover this
    HMC_NULL           = 3'b010, //cover this
    HMC_TS1_PART_ALIGN = 3'b011,
    HMC_TS1_FIND_REF   = 3'b100,
    HMC_TS1_ALIGN      = 3'b101,
    HMC_NULL_NEXT      = 3'b110,
    HMC_UP             = 3'b111  //cover this
  } status_init_rx_e;

  typedef enum bit [1:0] {
    INIT_TX_NULL_1 = 2'b00,
    INIT_TX_TS1    = 2'b01,
    INIT_TX_NULL_2 = 2'b10,
    INIT_DONE      = 2'b11
  } status_init_tx_e;

  `uvm_object_utils(reset4_seq)

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

  task body();
    string print_reg;
    super.body();

    setup_control();

    while(rf_rb.m_reg_status_init.status_init_rx_init_state.get() != HMC_UP) begin
      rf_rb.m_reg_status_init.read(status, data, .parent(this));
      if (rf_rb.m_reg_status_init.status_init_rx_init_state.get() == HMC_UP) begin
        print_reg = $sformatf("\n%s\n\tSTATUS INIT REGISTER\n%s\n\t status_init_rx_init_st
                              "*******************************","***************************
                              rf_rb.m_reg_status_init.status_init_rx_init_state.get(),rf_rb.r
                              ,"*****************************************************************
        `uvm_info("RESET_SEQ", print_reg,UVM_LOW)
        activate_reset();
      end
    end
```

Figure 6.26: example of a reset sequence

Chapter 7
AXI Agent

The AXI agent works as TX (connected to the TX link in DUT) and RX (connected to the RX link in DUT). It sends HMC request packets to Openhmc_top (DUT) and receives the response packets from it. It acts on behalf of user application. In this chapter, we are going to discuss the architecture of AXI agent components and the objects implemented in order to perform its task.

7.1 The AXI agent architecture

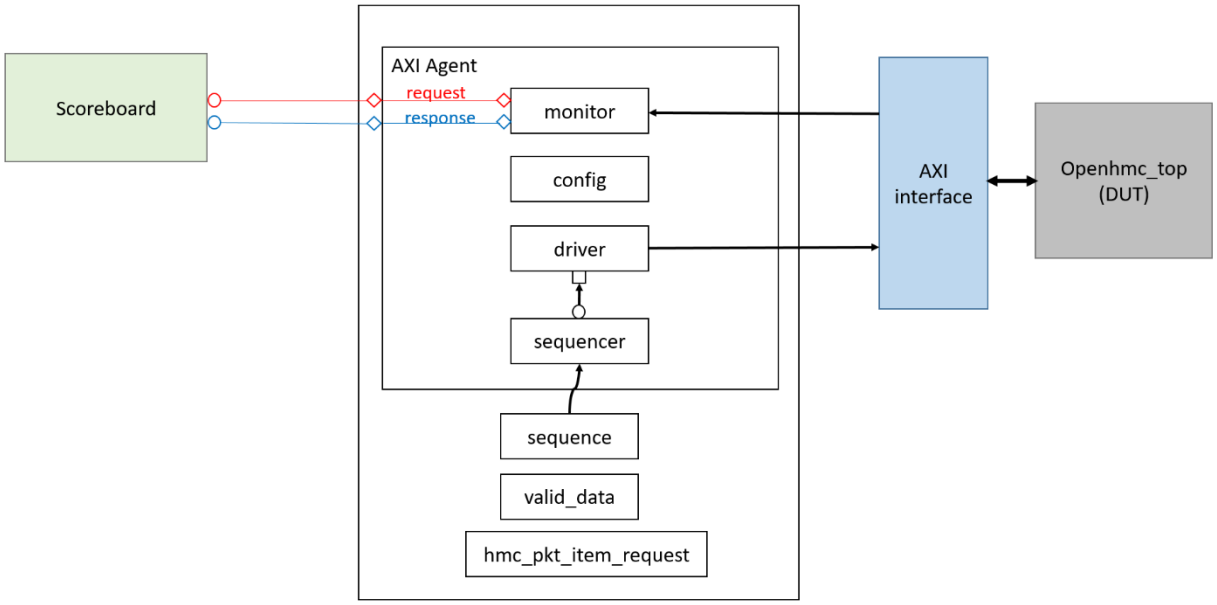


Figure 7.1: AXI Architecture

7.2 AXI agent components and other implemented objects

7.2.1 AXI interface

The interface is a major construct that is used to connect the design and testbench. It is a bundle of signals that aims to encapsulate the communication. AXI interface has *clk* and *res_n* as inputs to it. The frequency of the clock *clk* driving the AXI-4 interface must be equal to or higher than *clk_hmc*. For simplicity, we generated *clk* to be equal to *clk_hmc*.

AXI interface encapsulates the variables between the HMC controller and AXI driver and monitor. A brief description of these signals is shown in Table 7.1.

Table 7.1: AXI interface signals description

TX signal	Its direction w.r.t. AXI agent	Description
t_ready	Input	The HMC controller sets this flag when it is ready to receive new data/request packet.
t_valid	Output	The AXI driver sets this flag when it sends request packet. It allows the HMC controller to know that the data on t_data and t_user are valid and it can sample them when it is ready.
t_data	Output	The data sent by AXI agent. It is a 512-bit width. It consists of 4 flits, each flit is 128-bit width. This signal represents the request packet, part of it or multiple of them, depending on the length of the packets.
t_user	Output	It indicates whether each flit of t_data is valid, or if it is a header or a tail of a packet. Its length is 64bit, but only 12 bits is used. That is a bit for each flit.
RX signal	**Its direction w.r.t. AXI agent**	**Description**
rx_ready	Output	The AXI driver sets this flag when it is ready to receive new data/request packet from HMC controller. In the scope of our work, we set it always to one.
rx_valid	Input	The HMC controller sets this flag when it sends response packet. It allows the AXI monitor to know that the data on the rx_data and rx_user are valid and it can sample them and send them to the scoreboard.
rx_data	Input	The data sent by HMC controller. It is a 512-bit width. It consists of 4 flits, each flit is 128-bit width. This signal represents the response packet, part of it or multiple of them, depending on the length of the packets.
rx_user	Input	It indicates whether each flit of rx_data is valid, or if it is a header or a tail of a packet or if there is an error in it. Its length is 64bit, but only 16 bits is used. That is a bit for each flit.

7.2.2 hmc_pkt_item_request

This class extends *hmc_pkt_item* class. It is responsible for setting constraints on fields of HMC packet to match the specifications of a request packet. It is used by AXI sequence in order to generate the stimulus used by AXI driver. We set three constraints:

- Constraint to generate write, misc write, posted write, posted misc write, read and mode read commands.
- Constraint for tail fields to be equal zeros.
- Constraint to set the field is_ts1 to zero for correct pack.

7.2.3 valid_data

TDATA and *TUSER* are the data signals that are exchanged between the AXI agent and the DUT. This class encapsulates them to convert them from signal level to transaction level. *valid_data* is the sequence_item that the sequencer sends to the driver.

7.2.4 Sequence

The AXI sequence is responsible of generating the stimulus which is sent to the driver via the sequencer. It creates request packets of type *hmc_pkt_item_request* and converts them to *valid_data* items, as the latter is the sequence item exchanged between the sequencer and the driver. In one AXI cycle, the AXI agent can exchange with the DUT 512-bit TDATA and 64-bit TUSER, either transmitting using TX signals or receiving using RX signals. The request packet commands are of different lengths, they vary from 1 to 9 flits. Each flit is 128 bits. So in order to generate request packets and convert them to *valid_data* items the sequence has five methods does the following steps:

- It creates a number of new random request packets of type *hmc_pkt_item_request.*
- It assigns tags to the packets with commands that require tag in consecutive order. For other commands that do not require tag it sets tag field to zero. These commands are *posted write* and *posted misc write* commands.
- It divides the packets to flits and stores them in flit_queue.
- It assigns TUSER: header, tail, valid flags for each flit.
- It arranges the flits in axi_queue to form TDATA signal where each contains maximum 4 flits.
- It creates a *valid_data* item to each TDATA and TUSER.
- It sends the *valid_data* items to the sequencer.

7.2.5 Sequencer

It is the connection between the sequence and the driver. It takes the sequence item (*valid_data*) from the sequence and sends it to the driver through TLM Interface.

7.2.6 Config

This class configures the AXI agent as active agent. It sets the variable *agent_active* to be UVM_ACTIVE. This means that the AXI agent will contain a driver and a sequencer with the monitor.

7.2.7 Driver

It is the entity that drives the signals to the design via the interface handle. If the *res_n* signal is activated, it then sets *t_valid* to zero and *rx_ready* to one.
The driver is responsible for receiving the *valid_data* item from sequencer through TLM Interface between them, then it converts transaction level to signal level to drive the DUT. It does that by decapsulating *valid_data*: setting *t_data* to equal *valid_data* item's TDATA, and *t_user* to equal *valid_data* item's TUSER. As there is a valid data being sent, it sets the signal *t_valid* to one. The driver keeps the data on the buses without changing it until the DUT is ready to sample it. If *t_ready* is not set, it waits to the next positive edge clock and checks again.

7.2.8 Monitor

It is a passive entity that samples the DUT signals through the interface and converts signal level to transaction level. It is responsible for monitoring both input and output signals. It has two analysis ports connected to the scoreboard: one to write requests and the other to write responses.

Monitoring requests sent by AXI agent to DUT:

When the interface signals *t_valid* and *t_ready* equal one, it samples *t_data* and *t_user* and encapsulates them to a new *valid_data* item. The monitor has some methods to help restoring the request packet from va*lid_data* item.

First, the monitor divides 512-bit TDATA to 4 flits. Then it loops on each flit and checks TUSER valid flags to see if it is valid. If the flit is valid, it stores it in flit queue. Also it counts headers and tails seen in TDATA flits using TUSER header and tail flags.

The first flit in the flit queue will be header flit. It extracts the packet length from it. Then constructs a new packet of type *hmc_pkt_item* that has the same length and unpacks the flits to the packet. When completing the packet it stores it in *req_packet_queue*. Using the write method, the monitor sends the request packet to the scoreboard via the request analysis port.

Monitoring responses sent by DUT to AXI agent:

When the interface signal *rx_valid* equals one, it samples *rx_data* and *rx_user* and encapsulates them to a new *valid_data* item. As mentioned in table 6.1, we set *rx_ready* always to one so the monitor will have to check *rx_ready* only to know that the DUT has sent a response packet. The monitor has another methods to help restoring the response packet from *valid_data* item, which follow the same steps mentioned above. The difference is that the monitor stores the completed response packet in *res_packet_queue*. Using the write method, the monitor sends the response packet to the scoreboard via the response analysis port.

Chapter 8
HMC Agent

The HMC agent is a reactive agent which means that it receives requests from the dut and responds to them by sending responses back to the dut.

The agent contains 3 components which are:

1) Driver: Its role is to take what is sent from the dut through the HMC interface and send them into the sequence which responds based on what 's sent to it, thus the driver takes the response and sends it to the HMC interface again to be sent to the DUT.

2) Monitor: It receives packets sent from the HMC interface (as requests) or from the driver (as responses) and sends them to the scoreboard to be checked and to the coverage collector to calculate the coverage.

3) Sequencer: It connects between the sequences and the driver.

We can divide this agent into two stages occurring consequently and describes how each component mentioned above, along with the interface, works with these stages as follows:

8.1 Initialization Stage

8.1.1 driver_hmc_agent

It has a "for loop" with variable "l", this "for loop" is used to access each FLIT in the FPW (=4) at each positive edge of the clock. At the first entry to the driver, we create a request and a response packet sequence item of type "hmc_pkt_item". After that, at each value of "l" (representing the

```
Init status of the RX link
   3'b000: HMC_DOWN
   3'b001: HMC_WAIT_FOR_NULL
   3'b010: HMC_NULL
   3'b011: HMC_TS1_PART_ALIGN
   3'b100: HMC_TS1_FIND_REF
   3'b101: HMC_TS1_ALIGN
   3'b110: HMC_NULL_NEXT
   3'b111: HMC_UP
Init status of the TX link
   2'b00: INIT_TX_NULL_1
   2'b01: INIT_TX_TS1
   2'b10: INIT_TX_NULL_2
   2'b11: INIT_DONE
```

Figure 8.1: Tx and RX state of the DUT

FLIT position in the FPW, from 1 to 4), we check whether the tx state and the rx state of the dut are done doing initialization or not (at tx state=2'b11 and rx state=3'b111).

We get the tx and rx state of the dut through the the sequence called "hmc_state_seq", which is run through the virtual sequence called "hmc_init_vseq", by running the "rf_status_init_mirror_seq" firstly in the "hmc_init_vseq" to get the value of the tx and rx state from the register block connected to the register file in the dut, as mentioned in the RF-Agent chapter above. Through the "hmc_state_seq" which is run after the "rf_status_init_mirror_seq" in the initialization virtual sequence, the item sent to the "hmc_state_seq" has a field called "init_state" representing the tx state and a field called "rx_state" representing the rx state, these fields in the item are set to be equal to the tx and rx state in the register block accesed through the "rf_status_init_mirror_seq", this enables us to access the tx and rx state of the dut in the HMC driver without the need to access the register file itself to get these states and waste a lot of important clock cycles. After checking that the dut states is not done doing initialization, we enter the function called "packing_FLITS" once at each positive edge of the clock.

8.1.1.1 packing_FLITS() function

In this function, the driver gets the physical link from the dut to HMC interface (phy_data_tx_link2phy), which is used to send FLITs from the dut to the interface, and then the driver decides what type of FLITs is sent based on the LNG field bits that should be found in the first FLIT of any request or FLOW packet from bit number 10 to bit number 7 (4 bits), so if the LNG bits in the phy_data_tx_link2phy at the first FLIT in the FPW is not equal to zero then this is the first FLIT of a request and we got its length from LNG bits, but if LNG bits is equal to zero then it's either NULL FLIT or a training sequence, so the driver decides it's NULL FLIT when it checks that all the 128 bits of the FLIT is set to zeros, other than that, the driver decides it's a training sequence and raises the "is_TS1" flag by setting it to one. This process continues for all the FLITs in the FPW. After that, the driver has now the length of each request packet and their starting position in the FPW, if there are requests, and if it's NULL FLIT, the driver has the position of this FLIT in the FPW. Then, the driver pack the NULL FLIT in a dynamic allocated array.

8.1.1.2 NULL state

After getting out of "packing_FLITS" function, we check that the tx state field in the packet item is not equal to 2'b01 (INIT_TX_TS1 state) and that the variable "is_TS1" is set to 1'b0 which means that we are not in the training sequence state and that we are in either INIT_TX_NULL_1 or INIT_TX_NULL_2. After that, the driver will run "hmc_initialization_seq" sequence and send the response packet item to this sequence so that the response packet is set to be randomized with the command field equal to "NULL", then the driver packs the response packet item in a dynamic allocated array called "current_response_packet", then the driver sends this array to HMC_interface to put the content of this array on the physical link from the interface to the dut (phy_data_rx_phy2link), thus this is how the NULL FLITs are sent.

8.1.1.3 Training Sequence (TS1) state

After getting out of "packing_FLITS" function, we check that the tx state field in the packet item is equal to 2'b01 (INIT_TX_TS1 state) and that the variable "is_TS1" is set to 1'b1 which means that we are in the training sequence state. Then, at each positive edge of each clock cycle, a training sequence of the shown below is sent taking the full FPW and then the sequences are repeated each clock cycle but with some bits of it that change in orders as counters. When these counters are finished counting, they repeat their counts from the start again.

TS1 Ordering	Lane 0	Lane 1-14	Lane 15
0	16'hF030	16'hF050	16'hF0C0
1	16'hF031	16'hF051	16'hF0C1
2	16'hF032	16'hF052	16'hF0C2
3	16'hF033	16'hF053	16'hF0C3
...	...	...	...
14	16'hF03E	16'hF05E	16'hF0CE
15	16'hF03F	16'hF05F	16'hF0CF

Figure 8.2: TS1 Ordering at every positive edge of the clock cycle

8.1.2 monitor_hmc_agent

The monitor has two analysis ports, one for request packets and the other for response packets, and both ports are connected to their subscribers which are the scoreboard the coverage collector. The monitor receives the request and response packets from the HMC interface through the "notify_req_transaction" function for request packets and "notify_res_transaction" function for response packets. Both functions are declared in the monitor in the "run_phase" task and called in the interface file in the "run" task.

8.1.3 Sequences

8.1.3.1 hmc_init_vseq

It's a virtual sequence which runs the "rf_control_configuration_seq" sequence first, then runs the "rf_status_init_mirror_seq" sequence to get the dut states from the register block. So, we get the tx and rx states so that this virtual sequence will end at tx state =2'b11 and rx state =3'b110. Through this virtual sequence, we run the state sequence "hmc_state_seq". And after that, we check that the tx state is not equal to 2'b11 (INIT_DONE state) to run the initialization sequence "hmc_initialization_seq".

8.1.3.2 hmc_state_seq

This sequence is used to register the dut states into the states fields in the packet items used in the driver.

8.1.3.3 hmc_initialization_seq

This sequence is used to randomize response packet items with the command field equal to NULL (when the tx state is equal to either 2'b0 or 2'b10) and then send them back to the driver.

8.2 Request / Response Stage

In this stage, requests are sent from the dut through the HMC interface to the monitor of the hmc agent, this monitor is connected with analysis port to the driver of the hmc agent, thus when the monitor receives requests it sends them to the driver through the "write_request" function in the driver which pushes the request in a request buffer "request_queue" in the driver.

Each of the monitor and the driver has a handle to the link status class which enables the monitor to let the driver know the status or where the DUT stands in its status FSM so it follows it and send the transactions related to this status. The hmc driver, which is composed of "hmc_agent_base_driver" and "hmc_agent_driver", is the main component concerning the request/response stage, so we will talk about how it responds to the requests it receives.

8.2.1 link_up function

```
529   task hmc_agent_base_driver::link_up();
530      hmc_pkt_item packet;
531
532      get_packets(); // get response packets from the sequence
533      if (packet_queue.size() > 0 ) begin
534         packet = packet_queue.pop_front();  //-- send the first packet in the queue
535         if (next_poisoned < hmc_agent_cfg.poisoned_probability) begin //normal or poisoned?
536            send_poisoned(packet);
537         end else begin //else of line497
538            send_packet(packet);
539         end //end of line499
540         poisoned_propability_randomisation : assert (std::randomize(next_poisoned) with {next_poisoned > 0 && next_poisoned < 1000;});
541      end else if ($time-last_packet_timestamp > hmc_agent_cfg.send_pret_time) begin //else of line495
542         `uvm_info("HMC_AGENT_BASE_DRIVER_link_up()",$sformatf("sending pret, frp_queue size = %0d", frp_queue.size()), UVM_HIGH)
543         send_pret();
544      end else if ($time-last_packet_timestamp > hmc_agent_cfg.send_tret_time ) begin //else of line497
545         `uvm_info("HMC_AGENT_BASE_DRIVER_link_up()",$sformatf("sending tret, (<%0d)", used_tokens-sent_tokens), UVM_LOW)
546         send_tret();
547      end else begin //else of line499
548         `uvm_info("HMC_AGENT_BASE_DRIVER_link_up()",$sformatf("drive empty filt"),UVM_HIGH)
549         drive_flit(128'h0);
550      end
551
552      // From here on, there are no packets being driven.
553      // This is just logic to decide the next state.
554
555      //-- Handle error_abort_mode on remote link (hmc_link_status class)
556      if(remote_status.get_error_abort_mode() && $time() - start_retry_timestamp > hmc_agent_cfg.retry_timeout_period) begin
557         next_state = START_RETRY_INIT;
558      end
559
560   endtask : link_up
```

Figure 8.3: link_up function in hmc_agent_base_driver

When the driver detects that the link is up it runs the "link_up" function which gets the response of the request packet in the request buffer by running the function "get_packets". After running "get_packets" function, the response buffer now has the response packet generated as a response to the first request packet taken from the request buffer, and therefore the "link_up" function checks that the response buffer is not empty, and then takes the first response packet in the response buffer to send it to the HMC interface through "send_packet" function. This operation happens if a probability variable called "next_poisoned" is more than or equal to a threshold probability called

"poisoned_probability" defined in the agent configuration. If the "next_poisoned" probability is less than "poisoned_probability" the response packet is send as a poisoned packet to the HMC interface through "send_poisoned" function. If the response buffer is empty, the "link_up" function checks if the time from the last response packet sent is more than retry pointer return "pret" time defined in the agent configuration, then it sends pret packet to the HMC interface through the "send_pret" function. If not, the "link_up" function checks if the time from the last response packet sent is more than the token return "tret" time defined in the agent configuration, then it sends tret packet to the HMC interface through the "send_tret" function. Otherwise, NULL FLITs are sent to the HMC interface through "drive_flit" function by putting in its argument (128'h0).

8.2.2 get_packets function

```
567   task hmc_agent_base_driver::get_packets();
568      hmc_pkt_item pkt;
569
570      // check if the request_queue is empty then send requests to the sequence and return the response
571         if( request_queue.size() > 0) begin
572            if( packet_queue.size() == 0) begin
573               // give the request to the a function to generate response
574               pkt = request_queue.pop_front();
575               get_response(pkt);
576            end
577         end
578   endtask : get_packets
```

Figure 8.4: get_packets function in hmc_agent_base_driver

This function checks that the response buffer "packet_queue" is empty and that the request buffer "request_queue" is not empty to take the first request (using pop_front()) in the request buffer and then sends it to the "get_response" function.

8.2.3 get_response function

```
584 //*************************************************************************
585 function void hmc_agent_base_driver::get_response(input hmc_pkt_item request);
586   hmc_pkt_item response;
587   int response_length = 1;
588   //int new_timestamp;
589   //rand bit error_response;
590   bit [127:0] rand_flit;
591   bit [127:0] payload_flits [$];
592
593   `uvm_info("HMC_AGENT_BASE_DRIVER_get_response()",$sformatf("Generating response for a %s @%0x",request.command.name(), request.address),UVM_HIGH)
594   response = hmc_pkt_item::type_id::create("response");
595
596   //void'(this.randomize(error_response));
597   //new_timestamp = delay * 500ps + $time;
598
599   if (request.get_command_type() == READ_TYPE)
600   begin : read
601       // know the length
602       case (request.command)
603         RD16 : response_length = 2;
604         RD32 : response_length = 3;
605         RD48 : response_length = 4;
606         RD64 : response_length = 5;
607         RD80 : response_length = 6;
608         RD96 : response_length = 7;
609         RD112 : response_length = 8;
610         RD128 : response_length = 9;
611       endcase
612       // randomize the packet
613       void'(response.randomize() with {command == RD_RS;
614                                        address == request.address;
615                                        length  == response_length;
616                                        tag     == request.tag;
617                                       }); // error_status == 0 || error_response;
618       // randomize the packet's payload, maybe not needed?
619       for (int i = 0; i < response_length-1; i++) begin
620         randomize_flit_successful : assert(std::randomize(rand_flit));
621         payload_flits.push_front(rand_flit);
622       end
623       response.payload = payload_flits;
624       //response.timestamp = new_timestamp;
625       // enqueue the packet, ready to send
626       packet_queue.push_back(response);
627   end : read
628
629   else if (request.get_command_type() == MODE_READ_TYPE)
630    begin : mode_read
631       response_length = 2;
632       void'(response.randomize() with {command == MD_RD_RS;
633                                        address == request.address;
634                                        length  == response_length;
635                                        tag     == request.tag;
636                                       });
637       for (int i = 0; i < response_length-1; i++) begin
638         rand_flit = 128'h0; // mode read response has zeros in payload
```

Figure 8.5: get_response function in hmc_agent_base_driver

This function is used to generate response packet to the request packet, that is taken from the request buffer, based on the command field in the request packet, and then puts it at the end (using push_back()) of the response buffer.

Chapter 9
Scoreboard & Coverage Collector

9.1 Scoreboard

The scoreboard contains checkers and verifies the functionality of the design. It receives all transactions collected by monitor through an analysis export. Scoreboard contains a predictor or a reference model and a comparator to verify the DUT functionality. This is not mandatory, and all checkers can be implemented in the scoreboard itself as well. There are various scoreboard architectures which can be used as per the testbench requirement.

9.1.1 Predictor / Reference Model

Once the scoreboard receives the data transactions, it can either perform calculations and predict the expected output or send this data to a predictor or a reference model and get the expected output.

A predictor is a component which mimics the DUT logic or functionality and based on the sampled inputs, it predicts the output i.e., expected output.

This is then compared with the actual output i.e., sampled output from the DUT.

The predictor component is extended from uvm_subscriber base class. A uvm_component does not have a built-in analysis port while a uvm_subscriber is an extended version with a built-in analysis implementation port named as analysis_export. Thus, this class provides an analysis export for receiving transactions from a connected analysis export. Any class extended from this class must define the write method to process the incoming transactions.

9.1.2 Comparator

The comparator compares the actual output transactions with the expected output transactions and reports the comparison failures (and success) as they are detected.

9.1.3 Connection between scoreboard, predictor and comparator

Here we can say that the scoreboard code is mostly just a wrapper that includes the necessary export-ports to capture data transactions from monitor, plus the predictor and comparator.

We can observe that the scoreboard contains an analysis export *axp*. Three connections are required here:

1. Connect *axp* to predictor's *analysis_export* – The built-in *analysis_export* is connected to the scoreboard's *axp* analysis export and is used to capture the transaction that is passed to the scoreboard from the analysis port on the monitor.

2. Connect predictor's analysis port and comparator's expected analysis export – expected output is sent to comparator for comparison.
3. Connect comparator's actual analysis export to scoreboard's analysis export – actual data transactions sent to comparator for comparison.

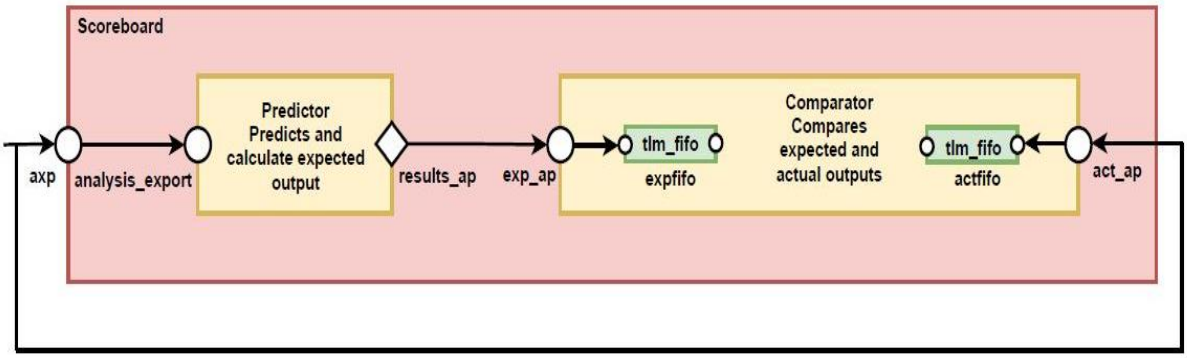


Figure 9.1: Connection between scoreboard, predictor, and comparator

9.1.4 UVM Scoreboard types

Depending on design functionality scoreboards can be implemented in two ways.

1. In-order scoreboard.
2. Out-of-order scoreboard.

9.1.4.1 In-order scoreboard

Performs a 1:1 element in order compares across all queues. Used to check that each secondary queue contains same contents as the primary queue, and that the ordering is exactly the same, also regarding the producer types. When a matching set is found, elements are removed from the respective queues. This will always be the first element of both primary and secondary queues. Error reporting: If the first element in a secondary queue is different from the first element in the primary queue, disregarding the producer type. Also, if a queue threshold/timeout happens.

To implement in order scoreboards, an easier way would be to implement TLM analysis FIFOs.

9.1.4.2 Out-of-order scoreboard

Performs a 1:1 element out of order compares across all queues. Used to check that each secondary queue contains the same contents as the primary queue, but without any specific ordering restrictions. When a matching set is found, elements are removed from the respective queues. Error reporting: If a queue grows above a set threshold, or a queue timeout happens.

We use out of order scoreboard in our design since the response packet might be delayed.

9.1.5 Scoreboard Functions

In our project we compare the transmitted packet with received packet and compare that the response packet with its expected response packet.

There are six functions that we have used in our scoreboard which are:

1. write_hmc_req.
2. write_hmc_rsp.
3. write_axi4_hmc_req.
4. write_axi4_hmc_rsp.
5. request_compare.
6. response_compare.

9.1.5.1 write_hmc_req function

When AXI4 send a request packet to hmc, the request packet stored in the AXI4 request queue and then when we want to write_hmc_req, we check if AXI4 request queue is empty. If it's empty, the scoreboard report that **"write_hmc_req: Unexpected packet (the request queue is empty)"**, else we compare the packet in queue with the exact packet "AXI4 send a request packet to hmc " and then scoreboard report the result of comparison.

9.1.5.2 write_hmc_rsp function

When AXI4 send a response packet to hmc, the response packet stored the in AXI4 response queue and then when we want to write_hmc_rsp. Then, we check that AXI4 response queue is empty. If it's empty, the scoreboard report that **"write_hmc_rsp: Unexpected packet (the response queue is empty)"**, else we compare the packet in queue with the exact packet "AXI4 send a response packet to hmc" and then scoreboard report the result of comparison.

9.1.5.3 write_axi4_hmc_req function

When hmc send a request packet to AXI4, the request packet stored the in (hmc) request queue and then when we want to write_ AXI4 _req. Then, we check if (hmc) request queue is empty. If it's empty, the scoreboard report **that "write_ AXI4 _req: Unexpected packet (the request queue is empty)"**, else we compare the packet in queue with the exact packet "hmc send a request packet to AXI4" and then scoreboard report the result of comparison.

9.1.5.4 write_axi4_hmc_rsp function

When hmc sends a response packet to AXI4, the response packet stored the in (hmc) response queue and then when we want to write_axi4_hmc_rsp. Then, we check that (AXI4) response queue is empty. If it's empty, the scoreboard report that **"write_axi4_hmc_rsp: Unexpected packet (the response queue is empty)"**, else we compare the packet in queue with the exact packet "hmc send a response packet to AXI4" and then scoreboard report the result of comparison.

9.1.5.5 request_compare function

We compare two request type packets and check all fields in the packets

- command
- cube_ID
- address
- tag
- payload

Then, finally report the result of comparison to know that where the error has happened.

9.1.5.6 response_compare function

We compare the received response packets with the expected response packet known from the request packet send by tag field which is unique for each request packet and finally report the result of comparison to know that where the error has happened.

9.1.6 report phase

The report() method is used to print the scoreboard information which are:

- Number of axi4_req_packet.
- Number of axi4_rsp_packet.
- Number of hmc_req_packet.
- Number of hmc_rsp_packet.
- Number of axi4_error_response.

9.2 Coverage Collector

Coverage is measure of the progress of a verification project in order to reduce the number of simulation cycles spent in verifying a design (indication of quality). Coverage is defined as the percentage of verification objectives that have been met. Types of coverage is presented in the following sections.

9.2.1 Code Coverage

- Tool generated.
- Automatically extracted from design code.
- Measures how much of the design code is exercised.
- Usually used to identify holes in the verification that has been performed.

What does 100 percent code coverage mean??

- The entire design implementation was executed.
- It does not provide an indication about the correctness of the verification suite.
- That doesn't mean the design is complete.
- What if the designer forgot to implement a feature from the specs? Would code coverage be affected?

9.2.1.1 Statement coverage

This type of code coverage means: Have all lines of code been executed?
Statement coverage can also be called block coverage or line coverage, where a block is a sequence of statements that are executed if a single statement is executed.

```
if (parity == ODD || parity == EVEN) begin
   tx <= compute_parity(data, parity);
   #(tx_time);
end
tx <= 1'b0;
#(tx_time);
if (stop_bits == 2) begin
   tx <= 1'b0;
   #(tx_time);
end
```

Figure 9.2: Statement Coverage Example

9.2.1.2 Branch coverage

This type of code coverage means: Have all branches in the code been taken?

9.2.1.3 Expression coverage

This type of code coverage means: Have all expressions that could affect a branch been executed? Expression coverage measures the various way decisions through the code.

```
if (parity == ODD || parity == EVEN) begin
   tx <= compute_parity(data, parity);
   #(tx_time);
end
tx <= 1'b0;
#(tx_time);
if (stop_bits == 2) begin
   tx <= 1'b0;
   #(tx_time);
end
```

Figure 9.3: Expression Coverage Example

9.2.1.4 State / Transition coverage

This type of code coverage means: Have all states of the state machine been active and have all transitions between states been traversed? This coverage has two types :

- State coverage: Which states have been visited.
- Transition coverage: did all possible transition happen.

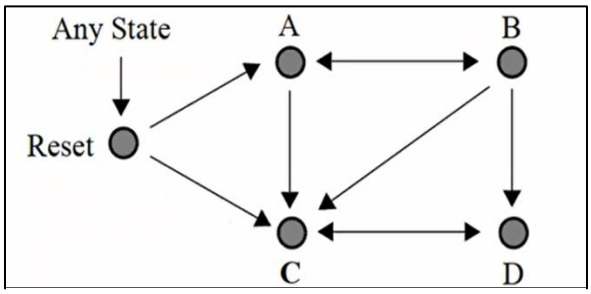


Figure 9.4: State/Transition Coverage Example

9.2.1.5 Toggle coverage

This type of code coverage means: Have all variables or all bits of all variables changed state and/or been through all transitions?

9.2.2 Functional coverage

Here are the properties and definition of the functional coverage

- Written according to feature from the specs.
- Functional coverage is another technology to help ensure that a bad design is not hiding behind passing testbenchs. (design tested well)
- Functional coverage records relevant metrics (packet length, instruction op_code, buffer occupancy level, etc.) to ensure that the verification process has exercised the design through all of the interesting values.
- Functional coverage is a user defined metric that measures how much of the design specification, as enumerated by features in the test plan, has been exercised.
- It can measure whether interested scenarios, corner cases, specification invariants, or other applicable design conditions have been observed, validate and tested.
- While code coverage measures how much of the implementation has been exercised, functional coverage measures how much of the original design specification has been exercised.
- High functional coverage does not necessarily correlate with high code coverage.

What does 100 percent functional coverage mean??

- It indicates completeness of the test suite, not correctness of design.
- Functional coverage indicates which interesting and relevant condition verified. Response to those conditions.
- Functional coverage metrics are only as good as the functional coverage model you have defined.

- Overall Functional coverage metric of 100 percent means that you have covered all of the coverage points you included in the simulation.

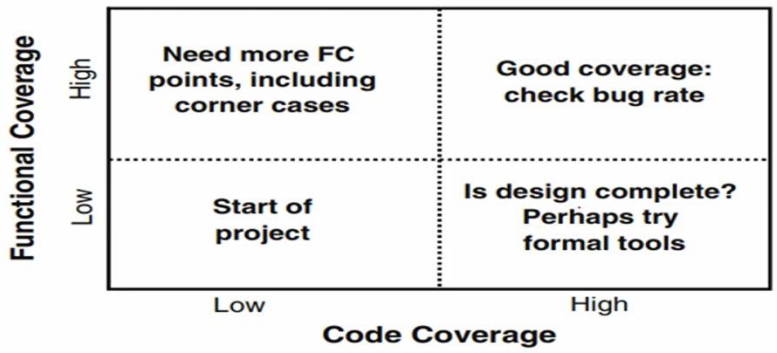


Figure 9.5: Coverage Comparison

9.2.2.1 Covergroups

The covergroup construct encapsulate the specification of a coverage model. Each covergroup specification can include the following components:

- A clocking event that synchronizes the sampling of the coverage points
- A set of coverage points
- Cross coverage between coverage points
- Optional formal arguments
- Coverage options

Some properties about cg creation:

- The covergroup construct is user_defined type. The type definition is written once, and multiple instances of that type can be created in different contexts.
- Similar to a class, once defined, a covergroup instance can be created via the new().
- A covergroup can be defined in a module, program, interface, or class.

The two major parts of covergroup are

- When the data is sampled “sample function”.
- What data values are sampled “defined inside cover group”.

Sampling can be done in two ways:

- Exactly calling <covergroup name>.sample() from procedural code.
- Using a control event.

A covergroup can contain one or more coverage points.

- Each coverage point can be associated with “bin”.
- On each sample clock simulator will increment the associated bin value.

9.2.2.2 Bins

Bins can be generated automatically. Some of the bin’s properties is presented below

- An automatically single bin will be created for each value of the cover point variable range.
- For an “n” bit integral cover point variable, a 2^n number of automatic bin will get created.
- Bins also be specified explicitly.
- A separate bin is created for each value in the given range of variable or a single/multiple bins for the range of values.
- Bins are explicitly declared within curly branches {} along with the bins keyword follow by bin name and variable value/range, immediately after the cover point identifier.
- Bins can also be specified for transitions.

9.2.2.3 Cross coverage points

Cross coverage is specified between the cover points or variables. Cross coverage is specified using the cross construct. Expression cannot be used directly in a cross; a coverage point must be explicitly defined first.

Chapter 10
Codes Summary

To implement the environment, we have written dozens of code files so we had to order them and make sure we name the files, classes and handles correctly and similar in all classes so we used a certain coding and ordering style from [8]. It is a great way to know where every type of class should be and build a good hierarchy of files that represents the actual hierarchy of the testbench environment.
First off, by taking the advice of the Si-Vision company supervisors, we have created a github repository which can be found in [9]. A github repo makes it easy to share code files and collaborate with each other and always track changes and become always up to date without having to always ask your teammates.

In this section, a small summary of the code files and repo layout will be presented.

10.1 Files ordering

Firstly, once you open the repo, you will see 4 folders as shown in (Figure 10.1). Following a brief explanation of the contents of each folder:

- Doc: contains any files of type word, pdf or powerpoint related to our own work and reports or related to the design and specs documentations
- RTL: here are the design codes of OpenHMC written in Verilog language
- Sim: here is everything related to the simulation, scripts that run the simulations and the output results of the simulation.
- Tb: all the files used in our testbench written by us.

Next, will go into details of simulation folder and its contents.

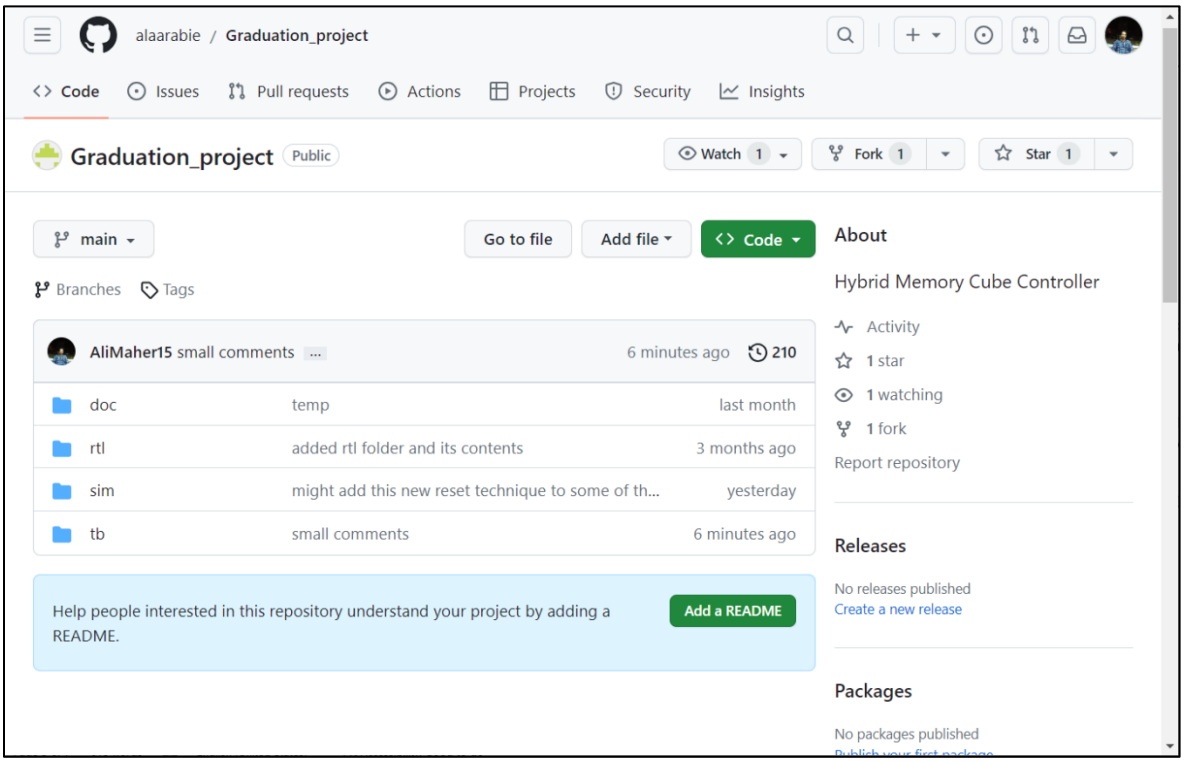


Figure 10.1: Github repo - Highest Level

10.1.1 Simulation Folder

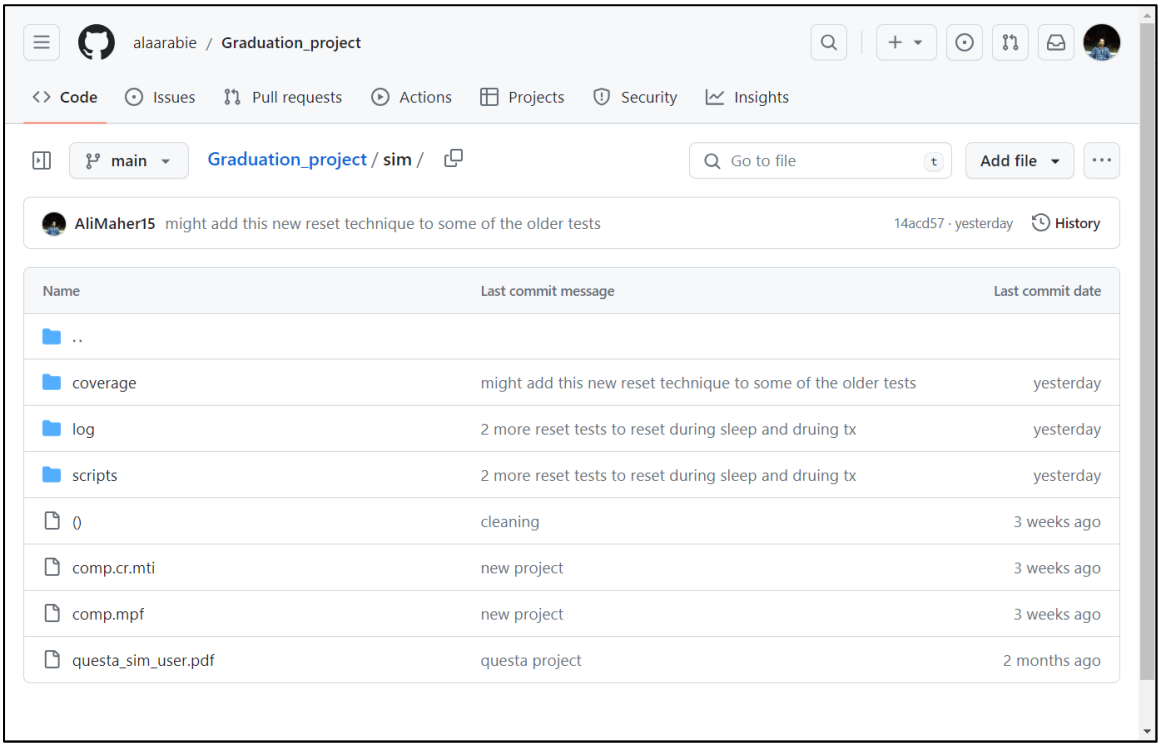


Figure 10.2: github simulation folder

The simulation folder contents as shown in (Figure 10.2) contains the main project file is called comp and it is the file of extension “mpf”. After running the simulation more files were generation automatically from questasim but to make the look of the folder cleaner only the important file of the project was uploaded to github and the other folders that contains the scripts we wrote and the results of running the simulation in the coverage folder as shown in (Figure 10.3).

main Graduation_project / sim / coverage / Go to file t Add file

AliMaher15 might add this new reset technique to some of the older tests 14acd57 · yesterday History

Name	Last commit message	Last commit date
..		
assertions.txt	might add this new reset technique to some of the older tests	yesterday
code_coverage.txt	might add this new reset technique to some of the older tests	yesterday
fun_coverage.txt	might add this new reset technique to some of the older tests	yesterday
hmc_retry_test.ucdb	HMC RTRY test	yesterday
openhmc.ucdb	2 more reset tests to reset during sleep and druing tx	yesterday
posted_only_test.ucdb	covered init fsm, hmc monitor isn't fully complete	2 days ago
read_only_test.ucdb	covered init fsm, hmc monitor isn't fully complete	2 days ago
reset1_test.ucdb	covered init fsm, hmc monitor isn't fully complete	2 days ago
reset2_test.ucdb	covered init fsm, hmc monitor isn't fully complete	2 days ago
reset3_test.ucdb	covered init fsm, hmc monitor isn't fully complete	2 days ago
reset4_test.ucdb	covered init fsm, hmc monitor isn't fully complete	2 days ago
reset5_test.ucdb	2 more reset tests to reset during sleep and druing tx	yesterday
reset_tests.ucdb	might add this new reset technique to some of the older tests	yesterday
simple_test.ucdb	covered init fsm, hmc monitor isn't fully complete	2 days ago
write_only_test.ucdb	covered init fsm, hmc monitor isn't fully complete	2 days ago

Figure 10.3: github coverage folder

If you notice there are two file types, first the ones with the extension “ucdb” which is short for Unified Coverage Database generated after running any test or simulation and it contains all the details about waves, coverage and assertions that happened during this simulation. The other type is text files which is generated by certain commands will be mentioned later to report the coverage results.

10.1.2 Testbench Folder

The folder as shown in (Figure 10.4) tried to follow [8] as much as we could to make a folder for each agent and sort the hierarchy of the testbench as the highest level is the testbench module and environment files, then under it by one level folder is all the components and files that are within the environment. In the sequence and test libraries we collected all used sequences, virtual sequences and tests in addition to their base classes files we used to cover the wanted scenarios.

Another approach for sequences is to add the sequence related to a specific agent inside the agent’s folder and not group all of them in the same folder like this.

main Graduation_project / tb / Go to file t Add file

AliMaher15 small comments 979fc8a · 32 minutes ago History

Name	Last commit message	Last commit date
..		
axi_agent	90% fun coverage 77% code coverage	last week
hmc_agent	small comments	32 minutes ago
hmc_packet	90% fun coverage 77% code coverage	last week
register_file_agent	trying to cover fsm transitions to reset	3 days ago
rf_reg_block	changing .sv to .svh for all classes	last week
seq_lib	small comments	32 minutes ago
test_lib	2 more reset tests to reset during sleep and druing tx	yesterday
coverage.svh	covered all initialization fsm by reset tests	2 days ago
env.svh	scoreboard rework is needed, and some cleanup	last week
env_cfg.svh	changing .sv to .svh for all classes	last week
openhmc_sva.sv	1- a way to enter sleep mode	3 days ago
scoreboard.svh	covered init fsm, hmc monitor isn't fully complete	2 days ago
system_if.sv	trying to cover fsm transitions to reset	3 days ago
tb_params_pkg.sv	1- a way to enter sleep mode	3 days ago
tb_pkg.sv	changing .sv to .svh for all classes	last week
tb_top.sv	covered all initialization fsm by reset tests	2 days ago
vsequencer.svh	changing .sv to .svh for all classes	last week

Figure 10.4: github testbench folder

If you notice there are two extension of the files the first is “sv” which is short for a system-verilog file the other is “svh” the h means header and it represents the classes that we won’t compile directly but we include them inside the packages and them compile the packages with all the files of type sv only and not svh. Next will show some examples of the contents of each of these folders

As shown in (Figure 10.6) the axi agent folder we can see that all the components related to the agent are started with a prefix axi_ to specify that this agent, driver, monitor, config object and so on are the ones of the axi agent because each agent will have this types of uvm classes and then all the files in this folder will be included inside the package file which will be compiled during simulation. To show more details here is how the package file looks like from the inside in (Figure 10.5).

```
package axi_pkg;
`include "uvm_macros.svh"
import uvm_pkg::*;
import cmd_pkg::*;

`include "axi_config.svh"
`include "valid_data.svh"
`include "hmc_pkt_item_request.svh"
`include "axi_sequencer.svh"
`include "axi_driver.svh"
`include "axi_monitor.svh"
`include "axi_agent.svh"


endpackage : axi_pkg
```

Figure 10.5: axi package file from inside

main Graduation_project / tb / axi_agent / Go to file t Add file

AliMaher15 90% fun coverage 77% code coverage def5705 · last week History

Name	Last commit message	Last commit date
..		
axi_agent.svh	changing .sv to .svh for all classes	last week
axi_config.svh	changing .sv to .svh for all classes	last week
axi_driver.svh	90% fun coverage 77% code coverage	last week
axi_interface.sv	renames, small edits, additions	2 weeks ago
axi_monitor.svh	90% fun coverage 77% code coverage	last week
axi_pkg.sv	changing .sv to .svh for all classes	last week
axi_sequencer.svh	changing .sv to .svh for all classes	last week
hmc_pkt_item_request.svh	changing .sv to .svh for all classes	last week
valid_data.svh	changing .sv to .svh for all classes	last week

Figure 10.6: github axi agent folder

Notice that the files inside the package have to be in a specific order in a down up way because if it is wrong the compilation will give errors when a handle to a class that wasn't compiled yet in the upper classes is mentioned so we have to start including from the lowest point.

The second example, is the register block folder shown in (Figure 10.7) here you can see how it is nicely order and named, for the classes that are extended from uvm_reg their names starts by a prefix reg_ and then because we only have one register block that contains all these register it is created and named a reg_block because it is extended from uvm_reg_block. And at the end all these files are included in the package.

main Graduation_project / tb / rf_reg_block / Go to file t Add file

AliMaher15 changing .sv to .svh for all classes 72f5d98 · last week History

Name	Last commit message	Last commit date
..		
reg_control.svh	changing .sv to .svh for all classes	last week
reg_counter_reset.svh	changing .sv to .svh for all classes	last week
reg_error_abort_not_cleared.svh	changing .sv to .svh for all classes	last week
reg_errors_on_rx.svh	changing .sv to .svh for all classes	last week
reg_poisoned_packets.svh	changing .sv to .svh for all classes	last week
reg_rcvd_rsp.svh	changing .sv to .svh for all classes	last week
reg_run_length_bit_flip.svh	changing .sv to .svh for all classes	last week
reg_sent_np.svh	changing .sv to .svh for all classes	last week
reg_sent_p.svh	changing .sv to .svh for all classes	last week
reg_sent_r.svh	changing .sv to .svh for all classes	last week
reg_status_general.svh	changing .sv to .svh for all classes	last week
reg_status_init.svh	changing .sv to .svh for all classes	last week
reg_tx_link_retries.svh	changing .sv to .svh for all classes	last week
rf_reg_block.svh	changing .sv to .svh for all classes	last week
rf_reg_block_pkg.sv	changing .sv to .svh for all classes	last week

Figure 10.7: github register block folder

Next is showing the sequence library in (Figure 10.8). There is one file that all the others extend from which is the base_seq file and it is originally extended from uvm_sequence base class, it contains variables and functions that would be repeated in each sequence so it is easily written only once. In addition, all files must end with _seq to know which uvm class are they. The sequence package contents can be shown in (Figure 10.9).

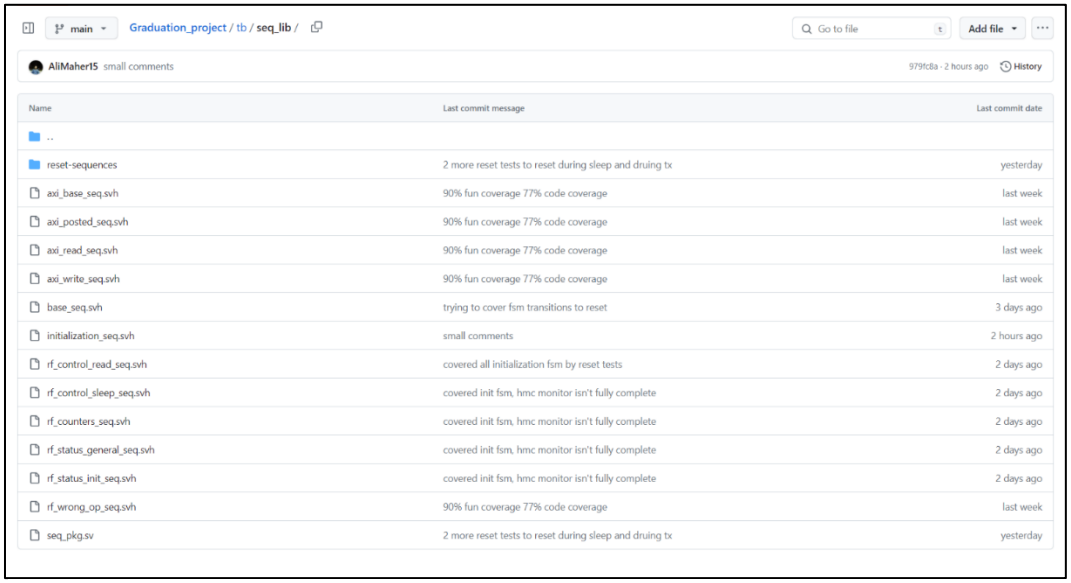


Figure 10.8: github sequence library folder

```
package seq_pkg ;
    import uvm_pkg::*;
    `include "uvm_macros.svh"
    import cmd_pkg::*;
    import axi_pkg::* ;
    import hmc_agent_pkg::* ;
    import rf_reg_block_pkg::* ;
    import rf_agent_pkg::* ;
    import tb_pkg::* ;


    `include "base_seq.svh"

    `include "rf_control_read_seq.svh"
    `include "rf_counters_seq.svh"
    `include "rf_status_general_seq.svh"
    `include "rf_control_sleep_seq.svh"
    `include "rf_status_init_seq.svh"
    `include "initialization_seq.svh"
    `include "rf_wrong_op_seq.svh"

    `include "axi_base_seq.svh"
    `include "axi_read_seq.svh"
    `include "axi_posted_seq.svh"
    `include "axi_write_seq.svh"

    `include "reset-sequences/reset1_seq.svh"
    `include "reset-sequences/reset2_seq.svh"
    `include "reset-sequences/reset3_seq.svh"
    `include "reset-sequences/reset4_seq.svh"
    `include "reset-sequences/reset5_seq.svh"


endpackage : seq_pkg
```

```
package test_pkg ;

  import uvm_pkg::*;
  `include "uvm_macros.svh"
  import cmd_pkg::*;
  import tb_params_pkg::* ;
  import axi_pkg::* ;
  import hmc_agent_pkg::* ;
  import rf_reg_block_pkg::* ;
  import rf_agent_pkg::* ;
  import seq_pkg::* ;
  import tb_pkg::* ;

  `include "vseq_base.svh"
  `include "base_test.svh"

  `include "simple-test/simple_test_vseq.svh"
  `include "simple-test/rf_check_vseq.svh"
  `include "simple-test/rf_wrong_op_vseq.svh"
  `include "simple-test/simple_test.svh"

  `include "read-only-test/read_only_test_vseq.svh"
  `include "read-only-test/read_only_test.svh"

  `include "write-only-test/write_only_test_vseq.svh"
  `include "write-only-test/write_only_test.svh"

  `include "posted-only-test/posted_only_test_vseq.svh"
  `include "posted-only-test/posted_only_test.svh"

  `include "hmc-retry-test/hmc_retry_test_vseq.svh"
  `include "hmc-retry-test/hmc_retry_test.svh"

  `include "reset-tests/reset1_test_vseq.svh"
  `include "reset-tests/reset1_test.svh"
  `include "reset-tests/reset2_test_vseq.svh"
  `include "reset-tests/reset2_test.svh"
  `include "reset-tests/reset3_test_vseq.svh"
  `include "reset-tests/reset3_test.svh"
  `include "reset-tests/reset4_test_vseq.svh"
  `include "reset-tests/reset4_test.svh"
  `include "reset-tests/reset5_test_vseq.svh"
  `include "reset-tests/reset5_test.svh"

endpackage : test_pkg
```

Figure 10.9: sequence and test packages

Finally, will show the test library folder in (Figure 10.10). The approach we took to run test is to first implement a virtual sequence which inside of it we can run multiple sequences that runs on multiple agents in the same time and it is a very helpful approach when we have more than one agent and interface in our environment.

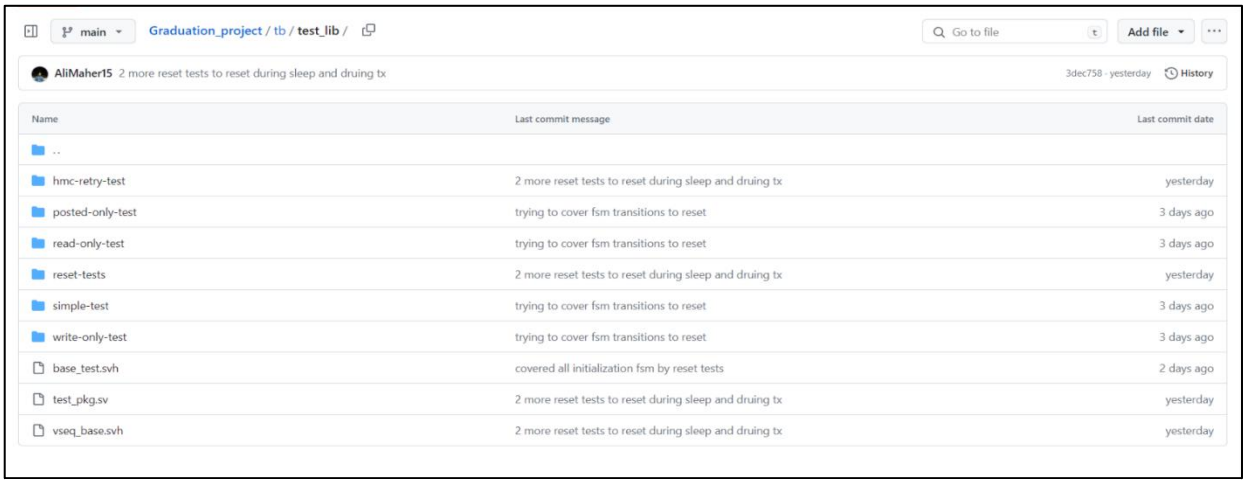


Figure 10.10: github test library folder

We separated each test in a single folder accompanied by its virtual sequence. The test package is shown in (Figure 10.9) we can see that we separated each test by one line for readability.

10.2 Statistics

Here will show some numbers of total files we wrote and how many lines of code in addition to some statistics from github.

Firstly, as shown in (Figure 10.11). Using an open-source tool called cloc or count line of code which we can pass a directory and it summarize what language and how many files and can even identify if the lines are coding or comments [10].

```
C:\count_line_of_code>cloc-1.96.exe tb --report-file=tb.txt
github.com/AlDanial/cloc v 1.96  T=0.12 s (831.3 files/s, 66815.2 lines/s)
-------------------------------------------------------------------------------
Language                         files          blank        comment           code
-------------------------------------------------------------------------------
Verilog-SystemVerilog              103           1602            895           5782
-------------------------------------------------------------------------------
SUM:                               103           1602            895           5782
-------------------------------------------------------------------------------
```

Figure 10.11: summary of lines of codes

So, for only the content of the testbench folder that we wrote all the files inside of it, they are a total of 103 files and 5782 lines of pure code. This shows the effort spent during this project.

Secondly, from the github side so far, we made more than 210 commits and the number can increase to add more cleaning in the codes and add more documentations.

A simple statistical analysis of all the edits since we started this repo can be presented by github and shown in (Figure 10.12). From it we can tell that we started working on the codes since April and kept working being active till the current day of writing this section. The inactive period in between is because of the final exams so we took a break and continued our work just after we were done. In total the time spent to reach our results is about 2 months.

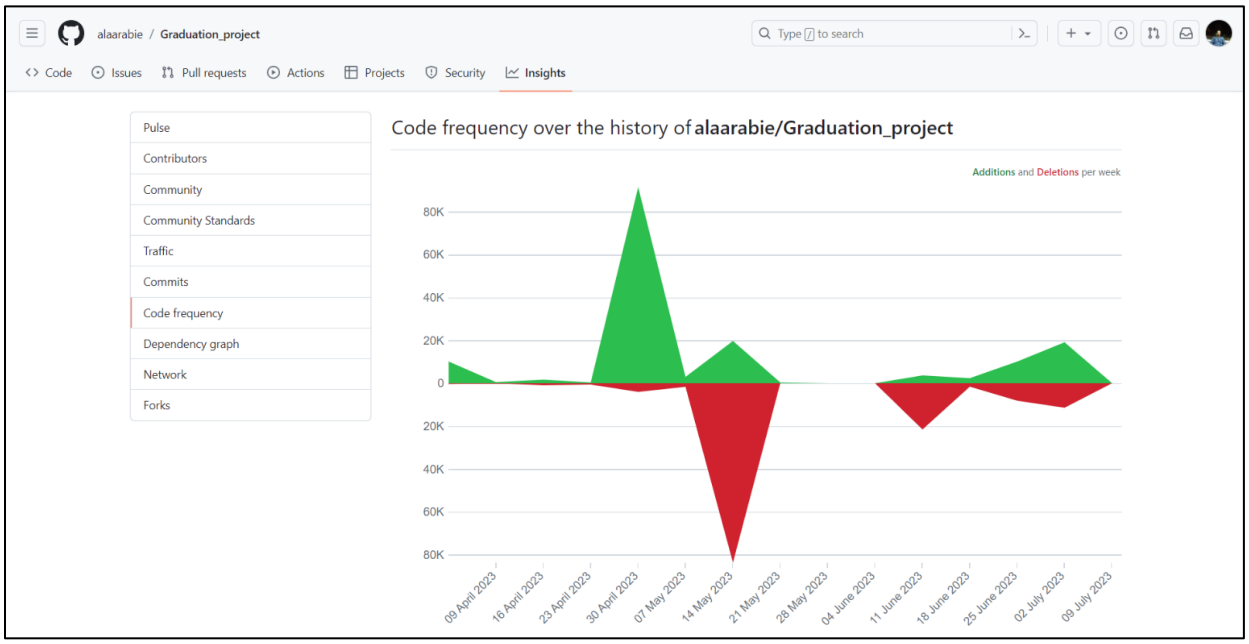


Figure 10.12: github code frequency

We wanted to check what is the average time to implement a uvm environment but all the results found that it varies from few weeks to few months as it depends on the complexity of the design, the experience level of the verification team, and the availability of reusable components. But as we knew nothing about verification before starting this project and had zero experience, we can tell by the results we reached that we did a fair job.

Another statistic shown the commit, addition and deletion during the time period of the repo in (Figure 10.13).

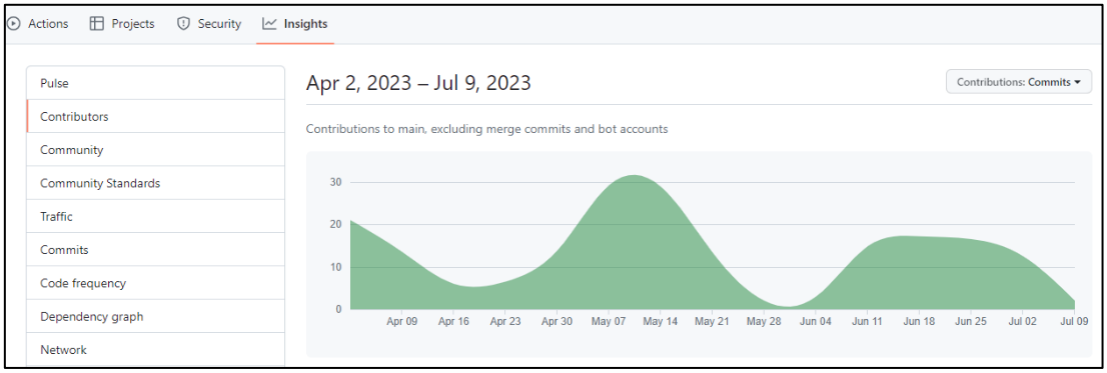


Figure 10.13: github contributions

10.3 Contents of Major Files

Here will show and explain briefly the role of some of the files to give a general idea how the environment works.

10.3.1 Top Level

First, like any testbench we have a testbench top module in it we make instances of all the interfaces and of the dut and connect the inputs and outputs of them between each other. Shown in (Figure 10.14) the instantiations of interfaces.

tb_top.sv

```
`timescale 100ps/1ps

module tb_top();

  `include "uvm_macros.svh"
  import uvm_pkg::*;
  import tb_params_pkg::*;
  import tb_pkg::* ;
  import seq_pkg::* ;
  import test_pkg::* ;

  parameter LANE_WIDTH = (FPW*128) / (2**LOG_NUM_LANES);

  logic clk;
  logic clk_hmc_refclk;

//**************** INTERFACES INSTANTIATIONS **************//
system_if system_if (.clk(clk));

  rf_if rf_if (.clk(clk), .res_n(system_if.res_n));

  hmc_agent_if #(.NUM_LANES(2**LOG_NUM_LANES)) hmc_if ();

  axi_interface #(.NUM_DATA_BYTES(FPW*16), .DWIDTH(FPW*128)) AXI_IF (.clk(clk), .res_n(system_if.res_n));

//*****************************************************//
```

Figure 10.14: testbench top and instantiation of interfaces

While in (Figure 10.15) a section of dut instantiation showing passing the parameters and connecting interfaces

```
tb_top.sv

//***************** DUT INSTANTIATION *****************//
  openhmc_top #(.FPW(FPW),
                .LOG_FPW(LOG_FPW),
                .LOG_NUM_LANES(LOG_NUM_LANES),
                .LOG_MAX_RX_TOKENS(LOG_MAX_RX_TOKENS),
                .LOG_MAX_HMC_TOKENS(LOG_MAX_HMC_TOKENS),
                .HMC_RX_AC_COUPLED(HMC_RX_AC_COUPLED),
                .DETECT_LANE_POLARITY(DETECT_LANE_POLARITY),
                .CTRL_LANE_POLARITY(CTRL_LANE_POLARITY),
                .CTRL_LANE_REVERSAL(CTRL_LANE_REVERSAL),
                .CTRL_SCRAMBLERS(CTRL_SCRAMBLERS),
                .OPEN_RSP_MODE(OPEN_RSP_MODE),
                .RX_RELAX_INIT_TIMING(RX_RELAX_INIT_TIMING),
                .RX_BIT_SLIP_CNT_LOG(RX_BIT_SLIP_CNT_LOG),
                .SYNC_AXI4_IF(SYNC_AXI4_IF),
                .XIL_CNT_PIPELINED(XIL_CNT_PIPELINED),
                .BITSLIP_SHIFT_RIGHT(BITSLIP_SHIFT_RIGHT),
                .DBG_RX_TOKEN_MON(DBG_RX_TOKEN_MON))

    dut (.clk_hmc(clk),
         .res_n_hmc(system_if.res_n),
         .res_n_user(system_if.res_n),
         .clk_user(clk),
         //axi interface
         .s_axis_tx_TVALID(AXI_IF.t_valid),
         .s_axis_tx_TREADY(AXI_IF.t_ready),
         .s_axis_tx_TDATA(AXI_IF.t_data),
         .s_axis_tx_TUSER(AXI_IF.t_user),
         .m_axis_rx_TVALID(AXI_IF.rx_valid),
         .m_axis_rx_TREADY(AXI_IF.rx_ready),
         .m_axis_rx_TDATA(AXI_IF.rx_data),
         .m_axis_rx_TUSER(AXI_IF.rx_user),
         // transceiver (physical link)
         .phy_data_tx_link2phy(to_serializers),
         .phy_data_rx_phy2link(from_deserializers),
         .phy_bit_slip(bit_slip), //Must be connected if DETECT_LANE_POLARITY==1 AND CTRL_LANE_POLARITY=0
         .phy_lane_polarity(phy_lane_polarity), //All 0 if CTRL_LANE_POLARITY=1
         .phy_tx_ready(system_if.res_n), //Optional information to RF
         .phy_rx_ready((phy_rx_ready===1) && LxTXPS), //Release RX descrambler reset when PHY ready
         .phy_init_cont_set(), //Can be used to release transceiver reset if used
```

Figure 10.15: how to instantiate the DUT

Finally, as shown in (Figure 10.16) an important step in the testbench top is to pass or set the handle to all the interfaces inside the uvm configuration database and specify which has access to it. As you notice it specified it to the uvm_test_top which is whatever uvm_test is running during simulation

```
tb_top.sv

initial begin
  uvm_config_db#(virtual rf_if)::set(null, "uvm_test_top", "rf_if", rf_if);

  uvm_config_db#(virtual hmc_agent_if #(.NUM_LANES(2**LOG_NUM_LANES)))::set(null, "uvm_test_top", "vif", hmc_if);

  uvm_config_db#(virtual system_if)::set(null, "uvm_test_top", "system_if", system_if);

  uvm_config_db#(virtual axi_interface #(.NUM_DATA_BYTES(FPW*16), .DWIDTH(FPW*128)))::set(null, "uvm_test_top", "AXI_IF", AXI_IF);
  run_test();
end
```

Figure 10.16: tb_top set the interfaces into uvm config db

10.3.2 Base Test

The layout of the file is shown in (Figure 10.17). It contains handles to all the configuration objects and interfaces the most important functions is the build_phase and the set_seqs function

The set_seqs function is responsible for when we start a virtual sequence it passes the handle to the environment config and the system interface to the sequence and the virtual sequence base also has a similar function to base these 2 handles to the sequences.

```
base_test.svh
class base_test extends  uvm_test;

  `uvm_component_utils(base_test)

  virtual system_if sys_if;

  axi_config_t m_axi_cfg;
  virtual axi_interface_t m_axi_if;

  hmc_agent_config_t m_hmc_cfg;
  virtual hmc_agent_if_t m_hmc_if;

  rf_agent_cfg m_rf_cfg;
  virtual rf_if m_rf_if;
  rf_reg_block rf_rb;

  env_cfg m_env_cfg;
  env m_env;


  function new(string name, uvm_component parent); ...
  endfunction : new


  function void build_phase(uvm_phase phase); ...
  endfunction : build_phase

  task run_phase(uvm_phase phase); ...
  endtask : run_phase


  function void set_seqs(vseq_base seq); ...
  endfunction

  // Print Testbench structure and factory contents
  function void start_of_simulation_phase(uvm_phase phase); ...
  endfunction : start_of_simulation_phase


endclass : base_test
```

Figure 10.17: base test file

For the build phase shown in (Figure 10.18). It basically creates the whole hierarchy of the environment by creating all the configuration objects and optionally configure any inner fields inside them if needed, and then getting the interfaces mentioned before in the testbench top from the configuration database and giving them to these configurations of each agent. Finally, create the environment configuration and the environment component.

```
base_test.svh
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    m_env_cfg = env_cfg::type_id::create ("m_env_cfg");
    m_hmc_cfg = hmc_agent_config_t::type_id::create("m_hmc_cfg") ;
    m_axi_cfg = axi_config_t::type_id::create("m_axi_cfg") ;
    rf_rb = rf_reg_block::type_id::create("rf_rb") ;
    rf_rb.build() ;
    m_rf_cfg = rf_agent_cfg::type_id::create("m_rf_cfg") ;

    if(!uvm_config_db #(hmc_agent_if_t)::get(this, "","vif",  m_hmc_cfg.vif))
    `uvm_fatal("TEST", "Failed to get hmc_if")
    if(!uvm_config_db #(axi_interface_t)::get(this, "","AXI_IF",  m_axi_cfg.vif))
    `uvm_fatal("TEST", "Failed to get axi_if")
    if(!uvm_config_db #(virtual rf_if)::get(this, "","rf_if", m_rf_cfg.vif))
    `uvm_fatal("TEST", "Failed to get rf_if")

    if(!uvm_config_db #(virtual system_if)::get(this, "","system_if", sys_if))
    `uvm_fatal("TEST", "Failed to get system_if")

    m_hmc_cfg.active=UVM_ACTIVE ;

    m_rf_cfg.active=UVM_ACTIVE ;

    m_axi_cfg.agent_active=UVM_ACTIVE ;

    m_env_cfg.rf_rb=rf_rb ;
    m_env_cfg.m_hmc_agent_cfg=m_hmc_cfg ;
    m_env_cfg.m_rf_agent_cfg=m_rf_cfg ;
    m_env_cfg.m_axi_cfg=m_axi_cfg;

    uvm_config_db #(env_cfg)::set(this, "*", "m_env_cfg", m_env_cfg);

    m_env = env::type_id::create("m_env",this);
  endfunction : build_phase
```

Figure 10.18: base test build phase

10.3.3 Environment

The general layout of the file is shown in (Figure 10.19). It consists of handles to each agent and the environment configuration object which contains inside of it all the other configurations for all the agents. Another component is the virtual sequencer which is a common way when we have multiple agents to implement a good hierarchy, we write a virtual sequencer and inside of it handles to the sequencers to all the agents under the environment so we can access them easier from the test.

```
env.svh
class env extends  uvm_env;

  `uvm_component_utils(env);

  vsequencer          m_vseqr;

  env_cfg             cfg;

  rf_agent          m_rf_agent;
  rf_reg2openhmc_adapter m_rf_reg2openhmc_adapter;
  uvm_reg_predictor#(rf_item) m_rf_openhmc2reg_predictor;

  hmc_agent_t hmc_agent_h ;

  axi_agent_t axi_agent_h;

  scoreboard m_scoreboard;
  coverage m_coverage;

  extern function new(string name, uvm_component parent);
  extern function void build_phase(uvm_phase phase);
  extern function void connect_phase(uvm_phase phase);

endclass : env
```

Figure 10.19: environment class layout

The approach taken to implement the virtual sequencer is explained in [11].

The build phase shown in (Figure 10.20) is exactly similar to the one in the base test as it does the same stuff. First get the configuration set by the base test above it, then create each component under it in the hierarchy and set the configurations required.

```
env.svh
function void env::build_phase(uvm_phase phase);
  super.build_phase(phase);

  if(!uvm_config_db #(env_cfg)::get(this, "","m_env_cfg", cfg))
    `uvm_fatal("ENV_CONFIG_LOAD", "Failed to get env_cfg from uvm_config_db")

  uvm_config_db#(rf_agent_cfg)::set(this, "m_rf_agent*", "rf_agent_cfg_t", cfg.m_rf_agent_cfg);
  uvm_config_db#(hmc_agent_config_t)::set(this, "hmc_agent_h*", "hmc_agent_config_t", cfg.m_hmc_agent_cfg);
  uvm_config_db#(axi_config_t)::set(this, "axi_agent_h*", "axi_config_t", cfg.m_axi_cfg);


  m_rf_agent = rf_agent::type_id::create("m_rf_agent",this);
  // Build the register model predictor
  m_rf_openhmc2reg_predictor = uvm_reg_predictor#(rf_item)::type_id::create("m_rf_openhmc2reg_predictor", this);
  m_rf_reg2openhmc_adapter = rf_reg2openhmc_adapter::type_id::create("m_rf_reg2openhmc_adapter");


  hmc_agent_h = hmc_agent_t::type_id::create("hmc_agent_h",this);

  axi_agent_h = axi_agent_t::type_id::create("axi_agent_h",this);

  m_vseqr    = vsequencer::type_id::create("m_vseqr" , this);

  m_scoreboard = scoreboard::type_id::create("m_scoreboard",this);
  m_coverage = coverage::type_id::create("m_coverage",this);

endfunction : build_phase
```

Figure 10.20: environment build phase

As for the connect phase shown in (Figure 10.21), its main purpose is to connect between analysis ports and imports mainly involving the monitors, scoreboard and coverage collector classes. But here the RAL has a specific code that has to be written in the connect phase of the environment to work.

```
env.svh
61  function void env::connect_phase(uvm_phase phase);
62    super.connect_phase(phase);
63
64    //*********************************************************************//
65    // Only set up register sequencer layering if the reg block is the top block
66    // If it isn't, then the top level environment will set up the correct sequencer
67    // and predictor
68    if(cfg.rf_rb.get_parent() == null) begin
69      `uvm_info("env_connect_phase","set up register block seqr", UVM_MEDIUM)
70      if(cfg.m_rf_agent_cfg.active == UVM_ACTIVE) begin
71        cfg.rf_rb.rf_map.set_sequencer(m_rf_agent.m_seqr, m_rf_reg2openhmc_adapter);
72      end
73      // Register prediction part:
74      // Set the predictor map:
75      m_rf_openhmc2reg_predictor.map = cfg.rf_rb.rf_map;
76      // Set the predictor adapter:
77      m_rf_openhmc2reg_predictor.adapter = m_rf_reg2openhmc_adapter;
78      // Disable the register models auto-prediction
79      cfg.rf_rb.rf_map.set_auto_predict(0);
80      // Connect the predictor to the bus agent monitor analysis port
81      m_rf_agent.rf_ap.connect(m_rf_openhmc2reg_predictor.bus_in);
82    end
83    //*********************************************************************//
84    m_vseqr.m_rf_seqr = m_rf_agent.m_seqr;
85    m_vseqr.m_hmc_sqr = hmc_agent_h.m_sqr;
86    m_vseqr.m_axi_sqr = axi_agent_h.a_sequencer;
87
88
89    // Connect Coverage and Scoreboard ports
90    axi_agent_h.mon_request.connect(m_coverage.analysis_export);
91    axi_agent_h.mon_response.connect(m_coverage.analysis_export);
92    hmc_agent_h.hmc_req_ap.connect(m_coverage.analysis_export);
93    hmc_agent_h.hmc_rsp_ap.connect(m_coverage.analysis_export);
94    hmc_agent_h.hmc_flow_ap.connect(m_coverage.analysis_export);
95
96    axi_agent_h.mon_request.connect(m_scoreboard.axi4_req_import);
97    axi_agent_h.mon_response.connect(m_scoreboard.axi4_rsp_import);
98    hmc_agent_h.hmc_req_ap.connect(m_scoreboard.hmc_req_import);
99    hmc_agent_h.hmc_rsp_ap.connect(m_scoreboard.hmc_rsp_import);
100
101 endfunction : connect_phase
```

Figure 10.21: environment connect phase

10.3.4 Agent

We have 3 agents in our environment, we will show as an example the file of the register file agent the layout is shown in (Figure 10.22).

rf_agent.svh

```
class rf_agent extends  uvm_agent;

  `uvm_component_param_utils(rf_agent )

  rf_agent_cfg  cfg;

  rf_driver   driver;
  rf_monitor  monitor;
  rf_sequencer   m_seqr;

  uvm_analysis_port #(rf_item) rf_ap;

  extern function new(string name, uvm_component parent);
  extern virtual function void build_phase(uvm_phase phase);
  extern virtual function void connect_phase(uvm_phase phase);


endclass : rf_agent
```

Figure 10.22: register file agent file layout

Similarly, to previous files in the build phase it creates all the lower components and in the connect phase it connects the driver with the sequencer and the analysis port inside the monitor with the outer analysis port inside the agent snapshot of these two functions is shown in (Figure 10.23)

rf_agent.svh

```
function void rf_agent::build_phase(uvm_phase phase);
  super.build_phase(phase);

  if(!uvm_config_db #(rf_agent_cfg)::get(this, "","rf_agent_cfg_t", cfg))
    `uvm_fatal("RF_AGENT_CONFIG_LOAD", "Failed to get rf_agent_cfg from uvm_config_db")

  if(cfg.active == UVM_ACTIVE) begin
      driver    = rf_driver::type_id::create("driver",this);
      driver.cfg = cfg;
      m_seqr = rf_sequencer::type_id::create("m_seqr", this);
  end

  monitor = rf_monitor::type_id::create("monitor",this);
  monitor.cfg = cfg;

  rf_ap = new("rf_ap",this);

endfunction : build_phase


function void rf_agent::connect_phase(uvm_phase phase);
  super.connect_phase(phase);

   if (cfg.active == UVM_ACTIVE) begin
    driver.seq_item_port.connect(m_seqr.seq_item_export);
  end
  monitor.rf_ap.connect(rf_ap);

endfunction : connect_phase
```

Figure 10.23: build and connect phase functions of rf agent

10.3.5 Parameters Package

Shown in (Figure 10.24). Since the DUT contains many parameters and can be configured in many ways, we must implement a way in our codes to control these parameters and configurations. So according to the UVM cookbook the way to do this is to make a package and only inside of it write the parameters [12].

tb_params_pkg.sv

```
package tb_params_pkg;

import axi_pkg::*;
import rf_agent_pkg::*;
import hmc_agent_pkg::*;

  parameter FPW                   = 4;    //Legal Values: 2,4,6,8
  parameter LOG_FPW               = 2;    //Legal Values: 1 for FPW=2 ,2 for FPW=4 ,3 for FPW=6/8
  //Define HMC interface width
  parameter LOG_NUM_LANES         = 4;    //Set 3 for half-width, 4 for full-width
  //Configure the Functionality
  parameter LOG_MAX_RX_TOKENS     = 10;   //Set the depth of the RX input buffer. Must be >= LOG(rf_rx_buffer_rtc) in the RF. Dont't care if OPEN_RSP_MODE=1
  parameter LOG_MAX_HMC_TOKENS    = 10;   //Set the depth of the HMC input buffer. Must be >= LOG of the corresponding field in the HMC internal register
  parameter HMC_RX_AC_COUPLED     = 1;    //Set to 0 to bypass the run length limiter, saves logic and 1 cycle delay
  parameter DETECT_LANE_POLARITY  = 1;    //Set to 0 if lane polarity is not applicable, saves logic
  parameter CTRL_LANE_POLARITY    = 1;    //Set to 0 if lane polarity is not applicable or performed by the transceivers, saves logic and 1 cycle delay
                                          //If set to 1: Only valid if DETECT_LANE_POLARITY==1, otherwise tied to zero
  parameter CTRL_LANE_REVERSAL    = 1;    //Set to 0 if lane reversal is not applicable or performed by the transceivers, saves logic
  parameter CTRL_SCRAMBLERS       = 1;    //Set to 0 to remove the option to disable (de-)scramblers for debugging, saves logic
  parameter OPEN_RSP_MODE         = 0;    //Set to 1 if running response open loop mode, bypasses the RX input buffer
  parameter RX_RELAX_INIT_TIMING  = 1;    //Per default, incoming TS1 sequences are only checked for the lane independent h'F0 sequence. Save resources and
                                          //eases timing closure. !Lane reversal is still detected
  parameter RX_BIT_SLIP_CNT_LOG   = 5;    //Define the number of cycles between bit slips. Refer to the transceiver user guide
                                          //Example: RX_BIT_SLIP_CNT_LOG=5 results in 2^5=32 cycles between two bit slips
  parameter SYNC_AXI4_IF          = 1;    //Set to 1 if AXI IF is synchronous to clk_hmc to use simple fifos
  parameter XIL_CNT_PIPELINED     = 1;    //If Xilinx counters are used, set to 1 to enabled output register pipelining
  //Set the direction of bitslip. Set to 1 if bitslip performs a shift right, otherwise set to 0 (see the corresponding transceiver user guide)
  parameter BITSLIP_SHIFT_RIGHT   = 1;
  //Debug Params
  parameter DBG_RX_TOKEN_MON      = 1;    //Set to 0 to remove the RX Link token monitor, saves logic


typedef hmc_agent_config #(.NUM_LANES(2**LOG_NUM_LANES)) hmc_agent_config_t;
typedef hmc_agent        #(.NUM_LANES(2**LOG_NUM_LANES)) hmc_agent_t;

typedef axi_config #(.NUM_DATA_BYTES(FPW*16), .DWIDTH(FPW*128)) axi_config_t;
typedef axi_agent  #(.NUM_DATA_BYTES(FPW*16), .DWIDTH(FPW*128)) axi_agent_t;
typedef axi_sequencer  #(.NUM_DATA_BYTES(FPW*16), .DWIDTH(FPW*128)) axi_sequencer_t;

typedef virtual hmc_agent_if #(.NUM_LANES(2**LOG_NUM_LANES)) hmc_agent_if_t;
typedef virtual axi_interface #(.NUM_DATA_BYTES(FPW*16), .DWIDTH(FPW*128)) axi_interface_t;

endpackage
```

Figure 10.24: content of parameters package

Chapter 11
Simulation and Results

In this chapter, we will focus mainly on the aspects of simulation and used tools while presenting the final results of simulation.

11.1 Tools

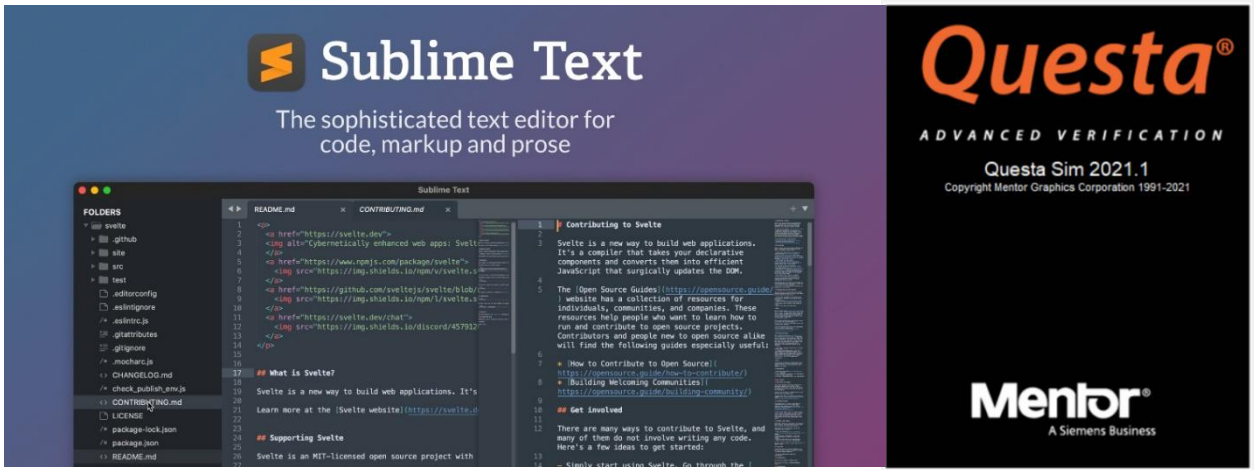


Figure 11.1: used tools

For functional verification the tools don't have to be very complicated and advanced. A little knowledge of the digital IC field will be enough to know how to work with these tools. Firstly, to write the codes and scripts all is required is an open-source text editor there are many great text editors but the editors that supports the systemverilog language are few. The text editor we used is called Sublime Text Editor 3 can be found at [13].

For the simulation, the used tool is Questasim. It is the easiest tool to access as it can be installed on windows and doesn't need a virtual machine. The version of questa was 2021.1 and the user manual of the tool followed to write the scripts can be found in [14].

The icons of the used tools are shown in (Figure 11.1).

11.2 Scripts

In this section, we will break down all the scripts written for the simulation. Meaning of the scripts that they are files of a language called TCL or Tool Command Language. It has some useful commands on its own as a language but in general for each tool we must read the user manual and each tool will have its own commands but they are common that they support this TCL language. [15]

The directory of the scripts and the seven scripts files written are shown in (Figure 11.2). The most important files that we will focus to explain are the run.do file and both dut.f and tb.f files. In addition to the waves file.

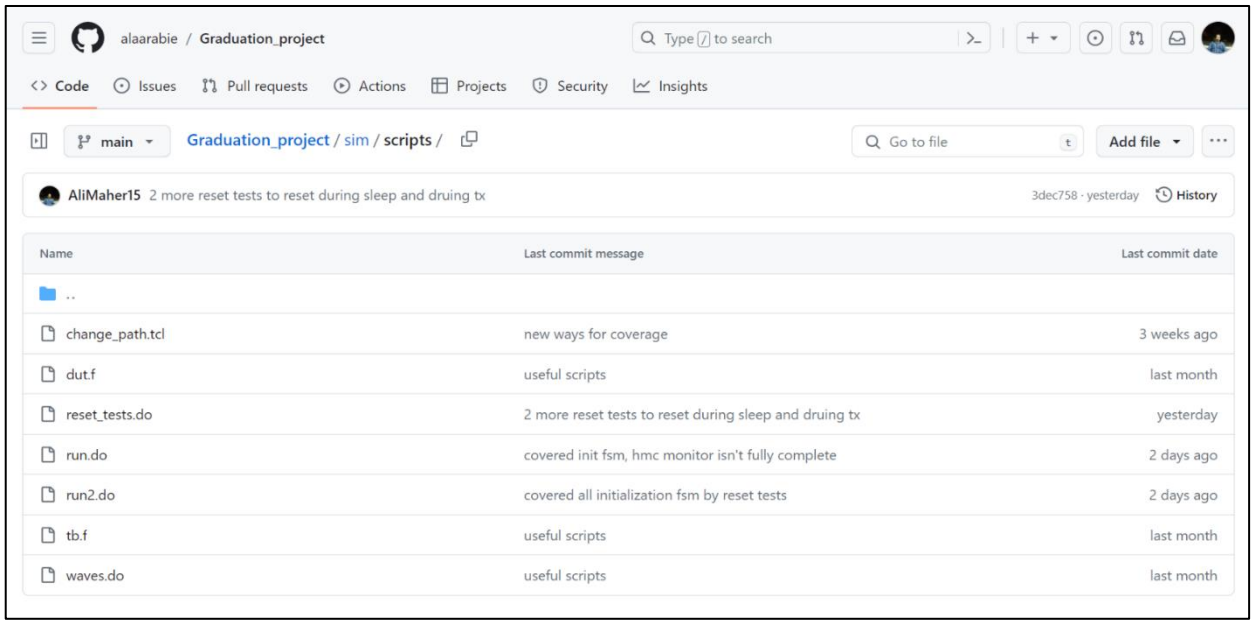


Figure 11.2: scripts directory

11.2.1 dut and tb files

As shown in (Figure 11.4) and (Figure 11.3). In these two files we list all the files that is needed to be compiled before running the test but because the way we are ordering our files that the simulation file that contains the questa project is parallel and in the same level of hierarchy to access the rtl or the testbench files we have to give it the whole path. To handle this while moving the files between use, the change_path.tcl file was created. It can generally be written in any language and its idea is to change the first part of the files path till the github repo using a code that does it automatically.

The part at the end of the two files that starts with +incdir means that these files might have include statements inside of them so for questasim to know the places of the files included we have to give it an include directory.

Also as mentioned before the files must be compiled in a certain order from down up to avoid the missing file error.

```
dut.f
D:/Ali/College/Graduation_project/rtl/building_blocks/counter/openhmc_counter48.v
D:/Ali/College/Graduation_project/rtl/building_blocks/counter/openhmc_counter48_wrapper_xilinx.v
D:/Ali/College/Graduation_project/rtl/building_blocks/fifos/async/openhmc_async_fifo.v
D:/Ali/College/Graduation_project/rtl/building_blocks/fifos/sync/openhmc_sync_fifo.v
D:/Ali/College/Graduation_project/rtl/building_blocks/fifos/sync/openhmc_sync_fifo_reg_based.v
D:/Ali/College/Graduation_project/rtl/building_blocks/fifos/sync/openhmc_sync_fifo_reg_stage.v
D:/Ali/College/Graduation_project/rtl/building_blocks/fifos/sync/xilinx/openhmc_srl_fifo_16.v
D:/Ali/College/Graduation_project/rtl/building_blocks/fifos/sync/xilinx/openhmc_sync_fifo_xilinx.v
D:/Ali/College/Graduation_project/rtl/building_blocks/rams/openhmc_ram.v
D:/Ali/College/Graduation_project/rtl/hmc_controller/crc/crc_128_init.v
D:/Ali/College/Graduation_project/rtl/hmc_controller/crc/crc_accu.v
D:/Ali/College/Graduation_project/rtl/hmc_controller/register_file/openhmc_rf.v
D:/Ali/College/Graduation_project/rtl/hmc_controller/rx/rx_crc_compare.v
D:/Ali/College/Graduation_project/rtl/hmc_controller/rx/rx_descrambler.v
D:/Ali/College/Graduation_project/rtl/hmc_controller/rx/rx_lane_logic.v
D:/Ali/College/Graduation_project/rtl/hmc_controller/rx/rx_link.v
D:/Ali/College/Graduation_project/rtl/hmc_controller/tx/tx_crc_combine.v
D:/Ali/College/Graduation_project/rtl/hmc_controller/tx/tx_run_length_limiter.v
D:/Ali/College/Graduation_project/rtl/hmc_controller/tx/tx_scrambler.v
D:/Ali/College/Graduation_project/rtl/hmc_controller/tx/tx_link.v
D:/Ali/College/Graduation_project/rtl/hmc_controller/openhmc_top.v



+incdir+D:/Ali/College/Graduation_project/rtl/include
```

Figure 11.3: dut script file

```
tb.f
D:/Ali/College/Graduation_project/tb/hmc_packet/cmd_pkg.sv

D:/Ali/College/Graduation_project/tb/rf_reg_block/rf_reg_block_pkg.sv

D:/Ali/College/Graduation_project/tb/register_file_agent/rf_agent_pkg.sv
D:/Ali/College/Graduation_project/tb/register_file_agent/rf_if.sv

D:/Ali/College/Graduation_project/tb/hmc_agent/hmc_agent_pkg.sv

D:/Ali/College/Graduation_project/tb/axi_agent/axi_pkg.sv
D:/Ali/College/Graduation_project/tb/axi_agent/axi_interface.sv

D:/Ali/College/Graduation_project/tb/tb_params_pkg.sv
D:/Ali/College/Graduation_project/tb/tb_pkg.sv

D:/Ali/College/Graduation_project/tb/hmc_agent/serdes/deserializer.v
D:/Ali/College/Graduation_project/tb/hmc_agent/serdes/serializer.v
D:/Ali/College/Graduation_project/tb/hmc_agent/hmc_agent_if.sv

D:/Ali/College/Graduation_project/tb/seq_lib/seq_pkg.sv
D:/Ali/College/Graduation_project/tb/test_lib/test_pkg.sv

D:/Ali/College/Graduation_project/tb/system_if.sv
D:/Ali/College/Graduation_project/tb/openhmc_sva.sv
D:/Ali/College/Graduation_project/tb/tb_top.sv


+incdir+D:/Ali/College/Graduation_project/tb/
+incdir+D:/Ali/College/Graduation_project/tb/axi_agent/
+incdir+D:/Ali/College/Graduation_project/tb/hmc_agent/
+incdir+D:/Ali/College/Graduation_project/tb/hmc_packet/
+incdir+D:/Ali/College/Graduation_project/tb/register_file_agent/
+incdir+D:/Ali/College/Graduation_project/tb/rf_reg_block/
+incdir+D:/Ali/College/Graduation_project/tb/seq_lib/
+incdir+D:/Ali/College/Graduation_project/tb/test_lib/
```

Figure 11.4: tb script file

11.2.2 waves file

Sometimes while running simulations debugging the waves is needed specially in the early stages in the coding when we aren't sure about the timing of our driving and monitoring and it is nice to check if we implemented a certain scenario on the waves and see how the signals react. So for that as shown in (Figure 11.5), a file that runs certain command to draw the pins of the DUT as well as the inner hmc interface.

waves.do

```
# Clocks
add wave /tb_top/dut/clk_hmc
add wave /tb_top/dut/res_n_hmc

# AXI
add wave -group AXI -color gold /tb_top/dut/s_axis_tx_TVALID \
                    /tb_top/dut/s_axis_tx_TREADY \
                    /tb_top/dut/s_axis_tx_TDATA \
                    /tb_top/dut/s_axis_tx_TUSER \
                    /tb_top/dut/m_axis_rx_TVALID \
                    /tb_top/dut/m_axis_rx_TREADY \
                    /tb_top/dut/m_axis_rx_TDATA \
                    /tb_top/dut/m_axis_rx_TUSER

# transceiver
add wave -group Transceiver -color Cyan  /tb_top/dut/phy_data_tx_link2phy \
                            /tb_top/dut/phy_data_rx_phy2link \
                            /tb_top/dut/phy_bit_slip \
                            /tb_top/dut/phy_lane_polarity \
                            /tb_top/dut/phy_tx_ready \
                            /tb_top/dut/phy_rx_ready \
                            /tb_top/dut/phy_init_cont_set

# hmc
add wave -group HMC -color White /tb_top/dut/P_RST_N \
                    /tb_top/dut/LXRXPS \
                    /tb_top/dut/LXTXPS \
                    /tb_top/dut/FERR_N

# Register file
add wave -group RF -color Magenta /tb_top/dut/rf_address \
                    /tb_top/dut/rf_read_data \
                    /tb_top/dut/rf_invalid_address \
                    /tb_top/dut/rf_access_complete \
                    /tb_top/dut/rf_read_en \
                    /tb_top/dut/rf_write_en \
                    /tb_top/dut/rf_write_data


# hmc_if
add wave -group HMC_IF -color Pink /tb_top/hmc_if/RXP \
                    /tb_top/hmc_if/RXN \
                    /tb_top/hmc_if/TXP \
                    /tb_top/hmc_if/TXN \
                    /tb_top/hmc_if/RXPS \
                    /tb_top/hmc_if/TXPS \
                    /tb_top/hmc_if/P_RST_N
```

Figure 11.5: waves script

11.2.3 Run file

The main file that is called to start simulation. We will show snapshots of the file and explain all its commands. The first snapshot shown in (Figure 11.6).

First part is as written from the comments, while compiling the files, questasim will automatically save a version of them inside a work library created by default. The files inside this library will then be used during simulation, but because we keep editing the files many times to avoid messing up anything we delete all the work library and create a new clean one.

Now for the transcript file, questasim can save its transcript inside a log file that shows everything that was printed during simulation, it is a nice feature to go back to this log if an error occurred or if we want to check printed results from the simulation.

Then the vlog command which is a command used to specifically compile Verilog files as you can tell from its name. We pass to it the dut and tb files shown before.

run.do

```
#*****************************************************#
# Clean Work Library
#*****************************************************#
if [file exists "work"] {vdel -all}
vlib work

#*****************************************************#
# Start a new Transcript File
#*****************************************************#
transcript file log/RUN_LOG.log
# better make one for each test

#*****************************************************#
# Compile RTL and TB files
#*****************************************************#
vlog -f scripts/dut.f
vlog -f scripts/tb.f

#*****************************************************#
# Optimizing Design with vopt
#*****************************************************#
vopt tb_top -o top_opt -debugdb  +acc +cover=sbecf+openhmc_top(rtl).
```

Figure 11.6: first part of run file

Now a very important command that must be used during simulation, and even if we didn't write it the questasim will do it but with default settings. The vopt command is explained in details in chapter 2 in the user manual [14]. We pass to it our testbench top module and the output will have a different name, if we didn't specify the output name, questasim will generate a default one but we have to specify it to use it later to run tests. Now the debugdb option enables some useful features that questa adds but it will take more time to optimize and an example of these features is that it can show a schematic of the module as blocks.

The other important option is the +cover. This one is responsible of specifying what types of code questasim will report for the code coverage as you notice we give it some letters but what does these letters mean?

Shown in (Figure 11.7) a snapshot of the user manual chapter 18 that each letter expresses a certain type of code.

So, to summarize the code coverage types we have chosen

- s: Statement
- b: Branch
- e: Expression
- c: Condition
- f: Finite State Machine (FSM)

Overview of Code Coverage Types

Questa SIM code coverage provides graphical and report file feedback on the following.

- **Statement coverage** — Counts the execution of each statement on a line individually, even if there are multiple statements in a line.
- **Branch coverage** — Counts the execution of each branch corresponding to if/else, ternary expressions, and conditional signal assignments, and indicates when a true or false condition has not executed.
- **Condition coverage** — Analyzes the decision made in "if" and ternary statements; can be considered an extension to branch coverage.
- **Expression coverage** — Analyzes expressions, for example those on the right hand side of assignment statements, and is similar to condition coverage.
- **Toggle coverage** — Counts each time a logic node transitions from one state to another.
- **FSM coverage** — Counts the states and transitions within a finite state machine.
- **SystemVerilog class coverage** — Collects data for each elaborated class type and each specialization of a parameterized class.

For details related to each of these types of coverage, see "Code Coverage Types".

Figure 11.7: snapshot from user manual of code coverage types

Now, to move to the second part of the run file shown in (Figure 11.8). This section is the one we run the tests so the blocks of the code a similar and the only difference is the test name each time.

run.do

```
#*****************************************************#
# Simulation of a Test
#*****************************************************#

#********************************** 1. Simple TEST ************************************#
transcript file log/simple_test.log
vsim top_opt -c -assertdebug -debugDB -fsmdebug -coverage +UVM_TESTNAME=simple_test
set NoQuitOnFinish 1
onbreak {resume}
log /* -r
run -all
coverage attribute -name TESTNAME -value simple_test
coverage save coverage/simple_test.ucdb

#********************************** 2. Read Only TEST ************************************#
transcript file log/read_only_test.log
vsim top_opt -c -assertdebug -debugDB -fsmdebug -coverage +UVM_TESTNAME=read_only_test
set NoQuitOnFinish 1
onbreak {resume}
log /* -r
run -all
coverage attribute -name TESTNAME -value read_only_test
coverage save coverage/read_only_test.ucdb

#********************************** 3. Posted Only TEST ************************************#
transcript file log/posted_only_test.log
vsim top_opt -c -assertdebug -debugDB -fsmdebug -coverage +UVM_TESTNAME=posted_only_test
set NoQuitOnFinish 1
onbreak {resume}
log /* -r
run -all
coverage attribute -name TESTNAME -value posted_only_test
coverage save coverage/posted_only_test.ucdb

#********************************** 4. Write Only TEST ************************************#
```

Figure 11.8: second snapshot of run file

We first open a new transcript that is saved into a new log file to go back to check what happened specifically during a test as the tests are stacked back-to-back and we won't have a chance to take a break in between to check the results. The code and functional coverage and all the details about the simulation is saved inside the UCDB file and we name it after the test name.

The vsim command is explained in chapter 6 of the user manual [14]. It has many many options but the ones we choose is to have the ability to draw the FSM as circles and show all the transitions and the other is to add to the code coverage the assertions and add assertions tab and we can also add any chosen assertion to the waves tab to check when the assertion started and ended.

The coverage command is explained in chapter 18 (code coverage) in the user manual [14] it has so many options but for the lack of time we chose the simplest method to use it.

The third part of the code shown in (Figure 11.9). Is after running all the tests we need to merge their outputs to know the summary of the results. By using the vcover merge command explained in chapter 18 code coverage in the user manual [14]. To merge everything into the output file named after the dut which is openhmc.ucdb and finally using the vcover report command to write all the coverage results inside text files and specifying what we need. Either the functional coverage results which are the covergroups we wrote inside our classes codes. Or the code coverage of statements, branches, FSMs. Or just a summary of all the assertions results and how many assertions we hit.

```
run.do

#*****************************************************#
# Close the Transcript file
#*****************************************************#
transcript file ()

#*****************************************************#
# draw the dut pins in waveforms
#*****************************************************#
#do waves.do

#*****************************************************#
# save the coverage in text files
#*****************************************************#
vcover merge  coverage/openhmc.ucdb \
              coverage/simple_test.ucdb \
              coverage/read_only_test.ucdb \
              coverage/posted_only_test.ucdb \
              coverage/write_only_test.ucdb \
              coverage/reset_tests.ucdb \
              coverage/hmc_retry_test.ucdb



vcover report coverage/openhmc.ucdb -cvg -details -output coverage/fun_coverage.txt
vcover report coverage/openhmc.ucdb -details -assert  -output coverage/assertions.txt
vcover report coverage/openhmc.ucdb  -output coverage/code_coverage.txt
```

Figure 11.9: last part of run file

11.3 Tests

In this section we will discuss each test and what was the intention while writing this test relating to either functional coverage or code coverage or even assertion coverage. We will start with the first test written and the one that runs the most virtual sequences and sequences.

11.3.1 Simple Test

simple_test.svh

```
class simple_test extends  base_test;
  `uvm_component_utils(simple_test)

 function new(string name, uvm_component parent);
    super.new(name,parent);
 endfunction : new

  task run_phase(uvm_phase phase);

    //vseq_class_name vseq_handle = seq_class_name::type_id::create("vseq_handle");
    rf_check_vseq m_rf_check_vseq = rf_check_vseq::type_id::create("m_rf_check_vseq");
    rf_wrong_op_vseq m_rf_wrong_op_vseq = rf_wrong_op_vseq::type_id::create("m_rf_wrong_op_vseq");
    simple_test_vseq m_simple_test_vseq = simple_test_vseq::type_id::create("m_simple_test_vseq");
    set_seqs(m_simple_test_vseq);
    set_seqs(m_rf_check_vseq);
    set_seqs(m_rf_wrong_op_vseq);

    phase.raise_objection(this);

    super.run_phase(phase);


      `uvm_info("SIMPLE_TEST","Starting test", UVM_MEDIUM)

      `uvm_info("SIMPLE_TEST","Executing simple_test_vseq", UVM_MEDIUM)
      m_simple_test_vseq.start(m_env.m_vseqr) ;
      `uvm_info("SIMPLE_TEST", "simple_test_vseq complete", UVM_MEDIUM)
      #1us
      `uvm_info("SIMPLE_TEST","Executing rf_check_vseq", UVM_MEDIUM)
      m_rf_check_vseq.start(m_env.m_vseqr);
      `uvm_info("SIMPLE_TEST", "rf_check_vseq complete", UVM_MEDIUM)
      #500ns
      `uvm_info("SIMPLE_TEST","Executing rf_wrong_op_vseq", UVM_MEDIUM)
      m_rf_wrong_op_vseq.start(m_env.m_vseqr);
      `uvm_info("SIMPLE_TEST", "rf_wrong_op_vseq complete", UVM_MEDIUM)
      #1us

      `uvm_info("SIMPLE_TEST","Ending test", UVM_MEDIUM)

    phase.drop_objection(this);

```

Figure 11.10: simple test file

The test file shown in (Figure 11.10). Runs three virtual sequences each one does specific operations. First it creates the sequences and gives them the configuration and system interface handles. Then it starts the main virtual sequence of the test called after the test name which is the simple_test_vseq. This virtual sequence will be presented next.

The objective of the simple test is to basically test the simple operation of the DUT as to send normal transactions without any errors and respond to it while following the standard operations as we must perform initialization at the start then wait for the idle state to send axi packets.

The objective was to just know if we wrote the codes correctly and the simulation can run without errors and after that we check the functional coverage it produced and what it needs then add more tests.

The virtual sequence as discussed before it can run multiple sequences that are directed to different agents. Shown in (Figure 11.11). First thing to do is the initialization sequence that must be run at the start of all tests without it the test will fail then after it enters a loop can be expressed on the fsm of the dut that it enters tx mode send axi packets then go to sleep then re initialize then send axi packets and so on.

simple_test_vseq.svh

```
class simple_test_vseq extends vseq_base ;

  `uvm_object_utils(simple_test_vseq)

  axi_base_seq axi_seq_h ;

  initialization_seq init;

  rf_control_sleep_seq sleep;


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

  task body();

    axi_seq_h=axi_base_seq::type_id::create("axi_seq_h") ;
    seq_set_cfg(axi_seq_h);
    init = initialization_seq::type_id::create("init");
    seq_set_cfg(init);
    sleep = rf_control_sleep_seq::type_id::create("sleep");
    seq_set_cfg(sleep);

    super.body();
      `uvm_info("simple_test_vseq", "Executing sequence", UVM_MEDIUM)
        init.start(m_rf_seqr);
        #2us;
        repeat(6) begin
          `uvm_info("simple_test_vseq", "Executing AXI sequence", UVM_MEDIUM)
          axi_seq_h.randomize();
          axi_seq_h.start(m_axi_sqr);
          `uvm_info("simple_test_vseq", "AXI sequence complete", UVM_MEDIUM)
          #5us;
          `uvm_info("simple_test_vseq", "START TRIAL - SET OpenHMC to Sleep then Re-Init", UVM_MEDIUM)
          sleep.start(m_rf_seqr);
          `uvm_info("simple_test_vseq", "END TRIAL - SET OpenHMC to Sleep then Re-Init", UVM_MEDIUM)
          #1us;
        end

  endtask : body
```

Figure 11.11: simple test virtual sequence

Shown in (Figure 11.12). How the virtual sequence and this test operates on the fsm of the dut.

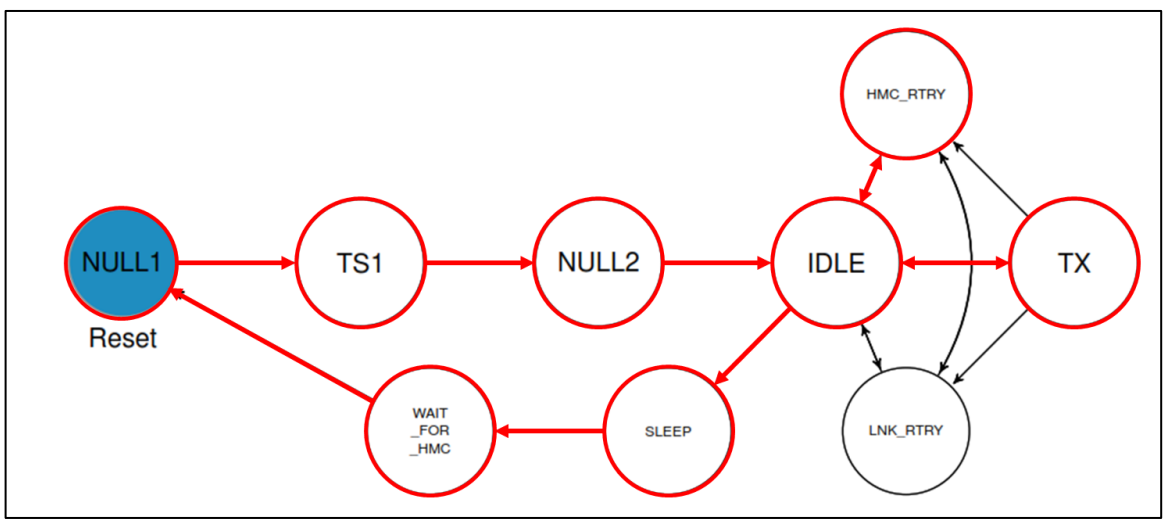


Figure 11.12: fsm during simple test

After a few loops of this a couple virtual sequences related to register file are run from the test file, the first one is called check virtual sequence and shown in (Figure 11.13).

rf_check_vseq.svh

```
class rf_check_vseq extends vseq_base ;

  `uvm_object_utils(rf_check_vseq)

  rf_control_read_seq rf_control_read_seq_h;
  rf_status_general_seq rf_status_general_seq_h;
  rf_status_init_seq rf_status_init_seq_h;
  rf_counters_seq rf_counters_seq_h;


  extern function new (string name = "");
  extern task body();

endclass : rf_check_vseq


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

task rf_check_vseq::body();

  rf_control_read_seq_h      = rf_control_read_seq::type_id::create("rf_control_read_seq_h");
  rf_status_general_seq_h    = rf_status_general_seq::type_id::create("rf_status_general_seq_h");
  rf_status_init_seq_h= rf_status_init_seq::type_id::create("rf_status_init_seq_h");
  rf_counters_seq_h= rf_counters_seq::type_id::create("rf_counters_seq_h");
  seq_set_cfg(rf_control_read_seq_h);
  seq_set_cfg(rf_status_general_seq_h);
  seq_set_cfg(rf_status_init_seq_h);
  seq_set_cfg(rf_counters_seq_h);

  super.body();

  `uvm_info("rf_check_vseq", "Executing sequence", UVM_MEDIUM)

      rf_status_general_seq_h.start(m_rf_seqr);
      rf_status_init_seq_h.start(m_rf_seqr);
      rf_control_read_seq_h.start(m_rf_seqr);
      rf_counters_seq_h.start(m_rf_seqr);

  `uvm_info("rf_check_vseq", "Sequence complete", UVM_MEDIUM)
```

Figure 11.13: rf check vseq

It performs read operations to go through all the registers of the dut and print it on the transcript.

The last virtual sequence is the one that intentionally perform wrong write or read operations on the register file to hit the assertion of invalid address pin.

After this test is run, we will present some of the simulation waves and transcript to show the operations mentioned in the actual simulation.

11.3.2 Simple Test Simulation

First thing we like to usually do in uvm is to print the whole environment hierarchy and all the components as shown in (Figure 11.14).

```
simple_test.log
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(278) @ 0: reporter [Questa UVM]
# UVM_INFO @ 0: reporter [RNTST] Running test simple_test...
# UVM_INFO D:/Ali/College/Graduation_project/tb//env.svh(69) @ 0: uvm_test_top.m_env [env_conne
# ---------------------------------------------------------------------------------
# Name                            Type                                Size  Value
# ---------------------------------------------------------------------------------
# uvm_test_top                    simple_test                         -     @490
#   m_env                         env                                 -     @574
#     axi_agent_h                 uvm_agent                           -     @677
#       a_driver                  uvm_driver #(REQ,RSP)               -     @850
#         rsp_port                uvm_analysis_port                   -     @865
#         seq_item_port           uvm_seq_item_pull_port              -     @857
#       a_monitor                 uvm_monitor                         -     @982
#         request                 uvm_analysis_port                   -     @1007
#         response                uvm_analysis_port                   -     @1015
#       a_sequencer               uvm_sequencer                       -     @873
#         rsp_export              uvm_analysis_export                 -     @880
#         seq_item_export         uvm_seq_item_pull_imp               -     @974
#         arbitration_queue       array                               0     -
#         lock_queue              array                               0     -
#         num_last_reqs           integral                            32    'd1
#         num_last_rsps           integral                            32    'd1
#       mon_request               uvm_analysis_port                   -     @989
#       mon_response              uvm_analysis_port                   -     @997
#     hmc_agent_h                 uvm_agent                           -     @625
#       h_status                  hmc_status                          -     @632
#         Requester_link_status   hmc_link_status                     -     @639
#         Responder_link_status   hmc_link_status                     -     @646
#       hmc_flow_ap               uvm_analysis_port                   -     @669
#       hmc_req_ap                uvm_analysis_port                   -     @653
#       hmc_rsp_ap                uvm_analysis_port                   -     @661
#       m_driver                  uvm_driver #(REQ,RSP)               -     @1246
#         flow_packets_port       uvm_analysis_port                   -     @1285
#         hmc_frp_port            uvm_analysis_imp_hmc_frp            -     @1269
#         request_import          uvm_analysis_imp_request            -     @1277
#         rsp_port                uvm_analysis_port                   -     @1261
#         seq_item_port           uvm_seq_item_pull_port              -     @1253
#       m_req_monitor             uvm_monitor                         -     @1029
#         frp_port                uvm_analysis_port                   -     @1052
#         item_collected_port     uvm_analysis_port                   -     @1036
#         req_cdr                 uvm_component                       -     @1297
```

Figure 11.14: snapshot of printing the testbench architecture

Next because the hmc agent could have multiple configurations. We print the configuration object of hmc agent as well despite not actually trying out these features as much in our tests, the printed configuration of hmc agent is shown in (Figure 11.15).

```
simple_test.log
####
#
# UVM_INFO D:/Ali/College/Graduation_project/tb/test_lib//base_test.svh(81) @ 0: uvm_test_top [BASE_TEST_
# ---------------------------------------------------------------
# Name                                  Type       Size  Value
# ---------------------------------------------------------------
# m_hmc_cfg                             <unknown>  -     @498
#   width                               integral   32    'd16
#   scramblers_enabled                  integral   1     'd0
#   reverse_polarity                    integral   16    'd0
#   reverse_lanes                       integral   1     'd0
#   run_length_limit                    integral   32    'd85
#   TS1_Messages                        integral   32    'd4
#   irtry_flit_count_to_send            integral   32    'h18
#   irtry_flit_count_received_threshold integral   32    'h10
#   hmc_tokens                          integral   32    'd163
#   rx_tokens                           integral   32    'd117
#   lane_errors_enabled                 integral   1     'd0
#   poisoned_probability                integral   32    'd0
#   lng_error_probability               integral   32    'd0
#   seq_error_probability               integral   32    'd0
#   crc_error_probability               integral   32    'd0
#   bitflip_error_probability           integral   32    'd0
# ---------------------------------------------------------------
```

Figure 11.15: printing configuration object

Now the test is ready to start, first thing is the initialization sequence and configure the dut from the control register. This part is shown in (Figure 11.16).

```
simple_test.log
271  # UVM_INFO D:/Ali/College/Graduation_project/tb/test_lib//simple-test/simple_test.svh(23) @ 502400: uvm_test_top [SIMPLE_TEST] Starting test
272  # UVM_INFO D:/Ali/College/Graduation_project/tb/test_lib//simple-test/simple_test.svh(25) @ 502400: uvm_test_top [SIMPLE_TEST] Executing simple_test_vseq
273  # UVM_INFO D:/Ali/College/Graduation_project/tb/test_lib//simple-test/simple_test_vseq.svh(26) @ 502400: uvm_test_top.m_env.m_vseqr@@m_simple_test_vseq [simple_test_vseq]
     Executing sequence
274  # UVM_INFO D:/Ali/College/Graduation_project/tb/seq_lib//initialization_seq.svh(23) @ 502400: uvm_test_top.m_env.m_rf_agent.m_seqr@@init [INITIALiZATION_SEQ] HMC_Token Count
     is:         163
275  # UVM_INFO D:/Ali/College/Graduation_project/tb/seq_lib//initialization_seq.svh(24) @ 502400: uvm_test_top.m_env.m_rf_agent.m_seqr@@init [INITIALiZATION_SEQ] RX_Token Count
     is:         117
276  # UVM_INFO D:/Ali/College/Graduation_project/tb/seq_lib//initialization_seq.svh(57) @ 2518401: uvm_test_top.m_env.m_rf_agent.m_seqr@@init [INITIALiZATION_SEQ]
277  # ********************************
278  #   CONTROL REGISTER
279  # ********************************
280  #    p_rst_n=0,
281  #    hmc_init_cont_set=0,
282  #    set_hmc_sleep=0,
283  #    warm_reset=0,
284  #    scrambler_disable=1,
285  #    run_length_enable=1,
286  #    rx_token_count=75,
287  #    irtry_received_threshold=10,
288  #    irtry_to_send=18
289  # ****************************************************************
```

Figure 11.16: configure the dut after starting test

Next the hmc monitor will print the status of each side the dut and the driver to keep us updated, the dut always starts the new state first then the driver follows it. Shown in (Figure 11.17) the printed messages.

```
simple_test.log
292  # UVM_INFO D:/Ali/College/Graduation_project/tb/seq_lib//initialization_seq.svh(71) @ 4540801: uvm_test_top.m_env.m_rf_agent.m_seqr@@init [INITIA
     the PHY TX to get ready
293  # UVM_INFO D:/Ali/College/Graduation_project/tb/seq_lib//initialization_seq.svh(73) @ 4540801: uvm_test_top.m_env.m_rf_agent.m_seqr@@init [INITIA
294  # UVM_INFO D:/Ali/College/Graduation_project/tb/hmc_agent//hmc_agent_monitor.svh(671) @ 4547200: uvm_test_top.m_env.hmc_agent_h.m_req_monitor [
     HMC_AGENT_MONITOR_link_states()] Requester Link current State: PRBS    vector: 110000
295  # UVM_INFO D:/Ali/College/Graduation_project/tb/hmc_agent//hmc_agent_monitor.svh(671) @ 4547200: uvm_test_top.m_env.hmc_agent_h.m_rsp_monitor [
     HMC_AGENT_MONITOR_link_states()] Responder Link current State: POWER_DOWN    vector: 100000
296  # UVM_INFO D:/Ali/College/Graduation_project/tb/hmc_agent//hmc_agent_monitor.svh(671) @ 4548650: uvm_test_top.m_env.hmc_agent_h.m_req_monitor [
     HMC_AGENT_MONITOR_link_states()] Requester Link current State: NULL_FLITS    vector: 111000
297  # UVM_INFO D:/Ali/College/Graduation_project/tb/hmc_agent//hmc_agent_monitor.svh(671) @ 5552000: uvm_test_top.m_env.hmc_agent_h.m_rsp_monitor [
     HMC_AGENT_MONITOR_link_states()] Responder Link current State: PRBS    vector: 110000
298  # UVM_INFO D:/Ali/College/Graduation_project/tb/hmc_agent//hmc_agent_monitor.svh(671) @ 5784450: uvm_test_top.m_env.hmc_agent_h.m_rsp_monitor [
     HMC_AGENT_MONITOR_link_states()] Responder Link current State: NULL_FLITS    vector: 111000
299  # UVM_INFO D:/Ali/College/Graduation_project/tb/hmc_agent//hmc_agent_monitor.svh(671) @ 6102450: uvm_test_top.m_env.hmc_agent_h.m_req_monitor [
     HMC_AGENT_MONITOR_link_states()] Requester Link current State: TS1    vector: 111100
300  # UVM_INFO D:/Ali/College/Graduation_project/tb/hmc_agent//hmc_agent_monitor.svh(390) @ 6108850: uvm_test_top.m_env.hmc_agent_h.m_req_monitor [
     HMC_AGENT_MONITOR_collect_flits()] Requester Link is not reversed
301  # UVM_INFO D:/Ali/College/Graduation_project/tb/seq_lib//initialization_seq.svh(87) @ 6553601: uvm_test_top.m_env.m_rf_agent.m_seqr@@init [INITIA
     PHY RX to get ready
302  # UVM_INFO D:/Ali/College/Graduation_project/tb/hmc_agent//hmc_agent_monitor.svh(671) @ 6814950: uvm_test_top.m_env.hmc_agent_h.m_rsp_monitor [
     HMC_AGENT_MONITOR_link_states()] Responder Link current State: TS1    vector: 111100
303  # UVM_INFO D:/Ali/College/Graduation_project/tb/hmc_agent//hmc_agent_monitor.svh(390) @ 6821350: uvm_test_top.m_env.hmc_agent_h.m_rsp_monitor [
     HMC_AGENT_MONITOR_collect_flits()] Responder Link is not reversed
304  # UVM_INFO D:/Ali/College/Graduation_project/tb/seq_lib//initialization_seq.svh(87) @ 7558401: uvm_test_top.m_env.m_rf_agent.m_seqr@@init [INITIA
     PHY RX to get ready
```

Figure 11.17: monitor printing states

And the rest of the states are shown in (Figure 11.18). Now once the link is up, we can send axi packets. If you notice in the figure that the status general register changed its content from training to link up and it is the field that is used to know we finished the initialization sequence and can continue.

```
simple_test.log
306  # ******************************
307  #   STATUS GENERAL REGISTER
308  # ******************************
309  #    link_up=0,
310  #    link_training=1,
311  #    sleep_mode=0,
312  #    FERR_N=1,
313  #    lanes_reversed=0,
314  #    phy_tx_ready=1,
315  #    phy_rx_ready=1,
316  #    hmc_tokens_remaining=0,
317  #    rx_tokens_remaining=75,
318  #    lane_polarity_reversed=0
319  # ************************************************************
320  #
321  # UVM_INFO D:/Ali/College/Graduation_project/tb/seq_lib//initialization_seq.svh(107) @ 7558401: uvm_test_top.m_env.m_rf_agent.m_seqr@@init [INIT
     ready
322  # UVM_INFO D:/Ali/College/Graduation_project/tb/hmc_agent//hmc_agent_monitor.svh(671) @ 7700850: uvm_test_top.m_env.hmc_agent_h.m_req_monitor [
     HMC_AGENT_MONITOR_link_states()] Requester Link current State: NULL_FLITS_2   vector: 111110
323  # UVM_INFO D:/Ali/College/Graduation_project/tb/hmc_agent//hmc_agent_monitor.svh(671) @ 7709350: uvm_test_top.m_env.hmc_agent_h.m_rsp_monitor [
     HMC_AGENT_MONITOR_link_states()] Responder Link current State: NULL_FLITS_2   vector: 111110
324  # UVM_INFO D:/Ali/College/Graduation_project/tb/hmc_agent//hmc_agent_monitor.svh(671) @ 7724850: uvm_test_top.m_env.hmc_agent_h.m_req_monitor [
     HMC_AGENT_MONITOR_link_states()] Requester Link current State: LINK_UP    vector: 111111
325  # UVM_INFO D:/Ali/College/Graduation_project/tb/hmc_agent//hmc_agent_monitor.svh(671) @ 7733350: uvm_test_top.m_env.hmc_agent_h.m_rsp_monitor [
     HMC_AGENT_MONITOR_link_states()] Responder Link current State: LINK_UP    vector: 111111
326  # UVM_INFO D:/Ali/College/Graduation_project/tb/seq_lib//initialization_seq.svh(135) @ 7740801: uvm_test_top.m_env.m_rf_agent.m_seqr@@init [INIT
327  # ******************************
328  #   STATUS GENERAL REGISTER
329  # ******************************
330  #    link_up=1,
331  #    link_training=0,
332  #    sleep_mode=0,
333  #    FERR_N=1,
334  #    lanes_reversed=0,
335  #    phy_tx_ready=1,
336  #    phy_rx_ready=1,
337  #    hmc_tokens_remaining=0,
338  #    rx_tokens_remaining=75,
339  #    lane_polarity_reversed=0
340  # ************************************************************
```

Figure 11.18: last part of states during initialization

The part we send axi packets now will be shown in (Figure 11.19). We send 5 random packets each time this packet is run. The command field is the field that is randomized and they are requests only because that's how the operation is supposed to be that axi sends requests and hmc sends responses.

```
simple_test.log
345  # UVM_INFO D:/Ali/College/Graduation_project/tb/test_lib//simple-test/simple_test_vseq.svh(30) @ 9740801: uvm_test_top.m_env.m_vseqr@@m_simple_test_vseq [simple_test_vseq]
     Executing AXI sequence
346  # UVM_INFO D:/Ali/College/Graduation_project/tb/seq_lib//axi_base_seq.svh(85) @ 9740801: uvm_test_top.m_env.axi_agent_h.a_sequencer@@axi_seq_h [axi_base_seq] HMC Packet #0
     command is: WR64 tag is: 1
347  # UVM_INFO D:/Ali/College/Graduation_project/tb/seq_lib//axi_base_seq.svh(85) @ 9740801: uvm_test_top.m_env.axi_agent_h.a_sequencer@@axi_seq_h [axi_base_seq] HMC Packet #1
     command is: P_WR48 tag is: 0
348  # UVM_INFO D:/Ali/College/Graduation_project/tb/seq_lib//axi_base_seq.svh(85) @ 9740801: uvm_test_top.m_env.axi_agent_h.a_sequencer@@axi_seq_h [axi_base_seq] HMC Packet #2
     command is: RD96 tag is: 1
349  # UVM_INFO D:/Ali/College/Graduation_project/tb/seq_lib//axi_base_seq.svh(85) @ 9740801: uvm_test_top.m_env.axi_agent_h.a_sequencer@@axi_seq_h [axi_base_seq] HMC Packet #3
     command is: WR64 tag is: 2
350  # UVM_INFO D:/Ali/College/Graduation_project/tb/seq_lib//axi_base_seq.svh(85) @ 9740801: uvm_test_top.m_env.axi_agent_h.a_sequencer@@axi_seq_h [axi_base_seq] HMC Packet #4
     command is: WR16 tag is: 3
351  # UVM_INFO D:/Ali/College/Graduation_project/tb/test_lib//simple-test/simple_test_vseq.svh(33) @ 9760000: uvm_test_top.m_env.m_vseqr@@m_simple_test_vseq [simple_test_vseq]
     AXI sequence complete
```

Figure 11.19: axi printing the created packets

In addition, we present how the axi part of the dut looks like during running this sequence in (Figure 11.20).

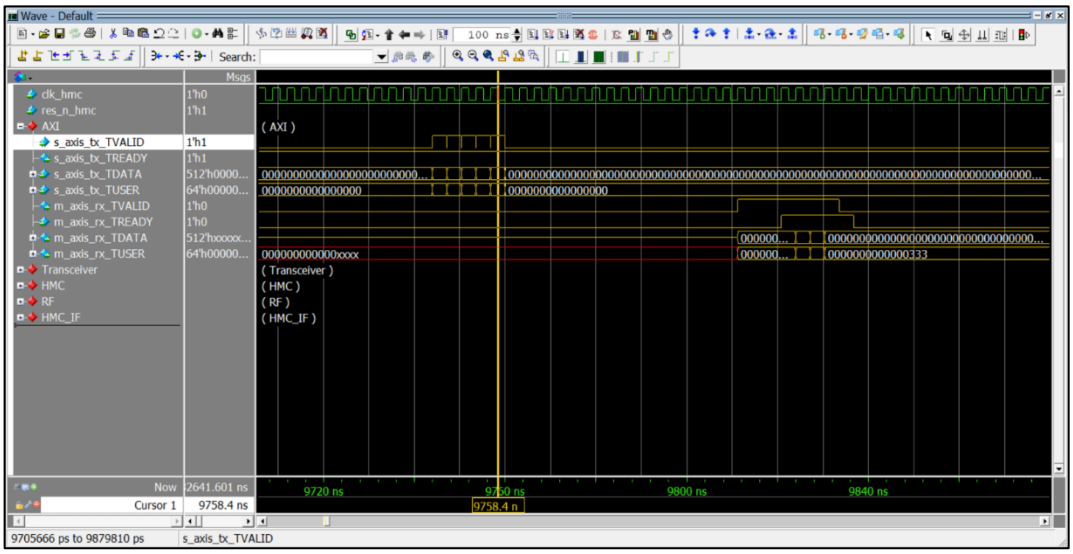


Figure 11.20: axi waves during axi sequence

Next is the sleep sequence and here are the printed messages in (Figure 11.21).

```
simple_test.log
352 # UVM_INFO D:/Ali/College/Graduation_project/tb/test_lib//simple-test/simple_test_vseq.svh(35) @ 14760000: uvm_test_top.m_env.m_vseqr@@m_simple_test_vseq [simple_test_vseq]
    START TRIAL - SET OpenHMC to Sleep then Re-Init
353 # UVM_INFO D:/Ali/College/Graduation_project/tb/seq_lib//rf_control_sleep_seq.svh(36) @ 14771201: uvm_test_top.m_env.m_rf_agent.m_seqr@@sleep [SLEEP_SEQ]
354 # ******************************
355 #   CONTROL REGISTER (before sleep)
356 # ******************************
357 #    p_rst_n=1,
358 #    hmc_init_cont_set=0,
359 #    set_hmc_sleep=0,
360 #    warm_reset=0,
361 #    scrambler_disable=1,
362 #    run_length_enable=1,
363 #    rx_token_count=75,
364 #    irtry_received_threshold=10,
365 #    irtry_to_send=18
366 # ****************************************************************
367 #
368 # UVM_INFO D:/Ali/College/Graduation_project/tb/hmc_agent//hmc_agent_monitor.svh(671) @ 14784000: uvm_test_top.m_env.hmc_agent_h.m_req_monitor [
    HMC_AGENT_MONITOR_link_states()] Requester Link current State: POWER_DOWN    vector: 101111
369 # UVM_INFO D:/Ali/College/Graduation_project/tb/seq_lib//rf_control_sleep_seq.svh(57) @ 14790401: uvm_test_top.m_env.m_rf_agent.m_seqr@@sleep [SLEEP_SEQ]
370 # ******************************
371 #   CONTROL REGISTER (after sleep)
372 # ******************************
373 #    p_rst_n=1,
374 #    hmc_init_cont_set=0,
375 #    set_hmc_sleep=1,
376 #    warm_reset=0,
377 #    scrambler_disable=1,
378 #    run_length_enable=1,
379 #    rx_token_count=75,
380 #    irtry_received_threshold=10,
381 #    irtry_to_send=18
382 # ****************************************************************
```

Figure 11.21: sleep sequences printed messages during simple test

Now it keeps repeating the same stuff for few loops then the 2 register virtual sequences activate, the wrong operations one doesn't print anything but the other check one prints some important information from the counters and shown in (Figure 11.22). Since this test isn't supposed to produce any errors and send all correct packets the numbers should match. So, it shows that it received zero poisoned packets and the number of non-poisoned packets received is equal to the number of non-posted requests sent so everything is working properly as intended.

```
simple_test.log
# ***************************************************************
#
# UVM_INFO D:/Ali/College/Graduation_project/tb/seq_lib//rf_counter
# ******************************
#   POSTED WRITE REQ COUNTER
# ******************************
#    sent_p=000000000000000e
# ***************************************************************
#
# UVM_INFO D:/Ali/College/Graduation_project/tb/seq_lib//rf_counter
# ******************************
#   READ REQ COUNTER
# ******************************
#    sent_r=0000000000000005
# ***************************************************************
#
# UVM_INFO D:/Ali/College/Graduation_project/tb/seq_lib//rf_counter
# ******************************
#   NON POSTED REQ COUNTER
# ******************************
#    sent_np=000000000000000f
# ***************************************************************
#
# UVM_INFO D:/Ali/College/Graduation_project/tb/seq_lib//rf_counter
# ******************************
#   POISONED PKTS RECEIVED COUNTER
# ******************************
#    poisoned_packets=0000000000000000
# ***************************************************************
#
# UVM_INFO D:/Ali/College/Graduation_project/tb/seq_lib//rf_counter
# ******************************
#   NON POISONED PKTS RECEIVED COUNTER
# ******************************
#    rcvd_rsp=000000000000000f
# ***************************************************************
```

Figure 11.22: register counters of packets

11.3.3 Simple Test Results

Here will show the initial code and functional coverage results.

Shown in (Figure 11.23) the initial functional coverage results of this test. Since everything is randomized some of the commands didn't get hit but for a 72% from first test it is a good result while mentioning also that some more coverage groups could be added but the current added groups are the only ones we intended to cover.

Covergroups

Name	Coverage	Goal	% of Goal	Status	Included
/tb_pkg/coverage	72.40%				
TYPE command_cg	49.16%	100	49.16%		✓
CVP command_cg::Flow	75.00%	100	75.00%		✓
CVP command_cg::Write_Requests	62.50%	100	62.50%		✓
CVP command_cg::Misc_Write_Requests	25.00%	100	25.00%		✓
bin Misc_Write_Requests[MD_WR]	1	1	100.00...		✓
bin Misc_Write_Requests[BWR]	0	1	0.00%		✓
bin Misc_Write_Requests[DUAL_2ADD8]	0	1	0.00%		✓
bin Misc_Write_Requests[SINGLE_ADD16]	0	1	0.00%		✓
CVP command_cg::Posted_Write_Requests	87.50%	100	87.50%		✓
CVP command_cg::Posted_Misc_Write_Requests	33.33%	100	33.33%		✓
bin Posted_Misc_Write_Requests[P_BWR]	0	1	0.00%		✓
bin Posted_Misc_Write_Requests[P_DUAL_2A...	1	1	100.00...		✓
bin Posted_Misc_Write_Requests[P_SINGLE_A...	0	1	0.00%		✓
CVP command_cg::Mode_Read_Request	0.00%	100	0.00%		✓
bin Mode_Read_Request[MD_RD]	0	1	0.00%		✓
CVP command_cg::Read_Requests	50.00%	100	50.00%		✓
CVP command_cg::Response_Commands	60.00%	100	60.00%		✓
TYPE length_cg	88.88%	100	88.88%		✓
TYPE cross_cg	79.16%	100	79.16%		✓

Figure 11.23: initial functional coverage from simple test

And to cover the rest of the commands some more tests that we increased the constrains and narrowed the randomization options to the commands that required covering.

Now, for the code coverage shown in (Figure 11.24) how the questasim user interface shows it.

Instance Coverage

Instance	Design unit	Design unit	Stmt %	Stmt graph	Branch %	Branch graph	State %	State graph	Transition %	Transition graph	FEC
/tb_top/dut	openhmc_top	Module	100.00%								
/tb_top/dut/ctrl_scramblers	openhmc_top	VlGenera...									
/tb_top/dut/openhmc_rf_I	openhmc_rf	Module	97.71%		97.50%						
/tb_top/dut/openhmc_rf_I/error_abort_not_cleared_...	openhmc_counter48	Module	75.00%		57.14%						
/tb_top/dut/openhmc_rf_I/errors_on_rx_count_I	openhmc_counter48	Module	75.00%		57.14%						
/tb_top/dut/openhmc_rf_I/run_length_bit_flip_count...	openhmc_counter48	Module	75.00%		57.14%						
/tb_top/dut/openhmc_rf_I/tx_link_retries_count_I	openhmc_counter48	Module	87.50%		71.43%						
/tb_top/dut/openhmc_sva_1	openhmc_sva	Module									
/tb_top/dut/regular_mode_use_rx_tokens	openhmc_top	VlGenera...									
/tb_top/dut/rx_link_I	rx_link	Module	97.21%		52.15%		100.00%		57.14%		
/tb_top/dut/rx_link_I/apply_lane_reversal[0]	rx_link	VlGenera...									
/tb_top/dut/rx_link_I/apply_lane_reversal[0]/genblk...		VlGenera...									
/tb_top/dut/rx_link_I/apply_lane_reversal[0]/genblk...		VlGenera...	100.00%		50.00%						
/tb_top/dut/rx_link_I/apply_lane_reversal[0]/genblk...		VlGenera...									
/tb_top/dut/rx_link_I/apply_lane_reversal[0]/genblk...		VlGenera...	100.00%		50.00%						
/tb_top/dut/rx_link_I/apply_lane_reversal[0]/genblk...		VlGenera...									
/tb_top/dut/rx_link_I/apply_lane_reversal[0]/genblk...		VlGenera...	100.00%		50.00%						
/tb_top/dut/rx_link_I/apply_lane_reversal[0]/genblk...		VlGenera...									
/tb_top/dut/rx_link_I/apply_lane_reversal[0]/genblk...		VlGenera...	100.00%		50.00%						
/tb_top/dut/rx_link_I/apply_lane_reversal[0]/genblk...		VlGenera...									
/tb_top/dut/rx_link_I/apply_lane_reversal[0]/genblk...		VlGenera...	100.00%		50.00%						
/tb_top/dut/rx_link_I/apply_lane_reversal[0]/genblk...		VlGenera...									
/tb_top/dut/rx_link_I/apply_lane_reversal[0]/genblk...		VlGenera...	100.00%		50.00%						
/tb_top/dut/rx_link_I/apply_lane_reversal[0]/genblk...		VlGenera...									

Figure 11.24: shape of initial code coverage after simple test

For a better summary we can use the vcover report command to check the total percentages. Shown in (Figure 11.25).

```
TOTAL COVERGROUP COVERAGE: 72.40%  COVERGROUP TYPES: 3

TOTAL ASSERTION COVERAGE: 92.30%  ASSERTIONS: 39

Total Coverage By Instance (filtered view): 76.47%
```

Figure 11.25: initial coverage report from simple test

So, from one test only and the first test to run we hit 76% code coverage despite missing out on some features. And that shows the benefits of the constrained random testing concept that is explained in a reference [1]. Shown in (Figure 11.26) the usual timetable for coverage.

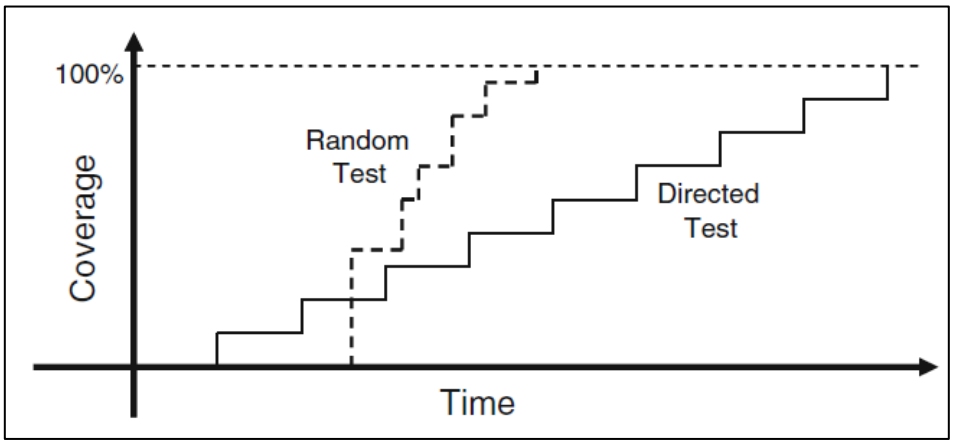


Figure 11.26: timetable of constrained random testing

Now what is left is the finite state machine coverage and we can look at it from questasim as it shows a good summary of all states and transitions. Shown in (Figure 11.27) all the finite state machines in the dut.

FSM List

Instance	States	Transitions
sim:/tb_top/dut/tx_link_I/init_state	4	6
sim:/tb_top/dut/tx_link_I/state	7	17
sim:/tb_top/dut/rx_link_I/rf_rx_init_status	8	14

Figure 11.27: FSMs of the DUT

In total they are 3 FSMs contains 19 states and 37 possible transitions. Next will show the coverage of each of these FSMs.

First and the main one is the state FSM which is the main one of the DUT the other two are only related to the initialization process. Shown in (Figure 11.28) the state FSM.

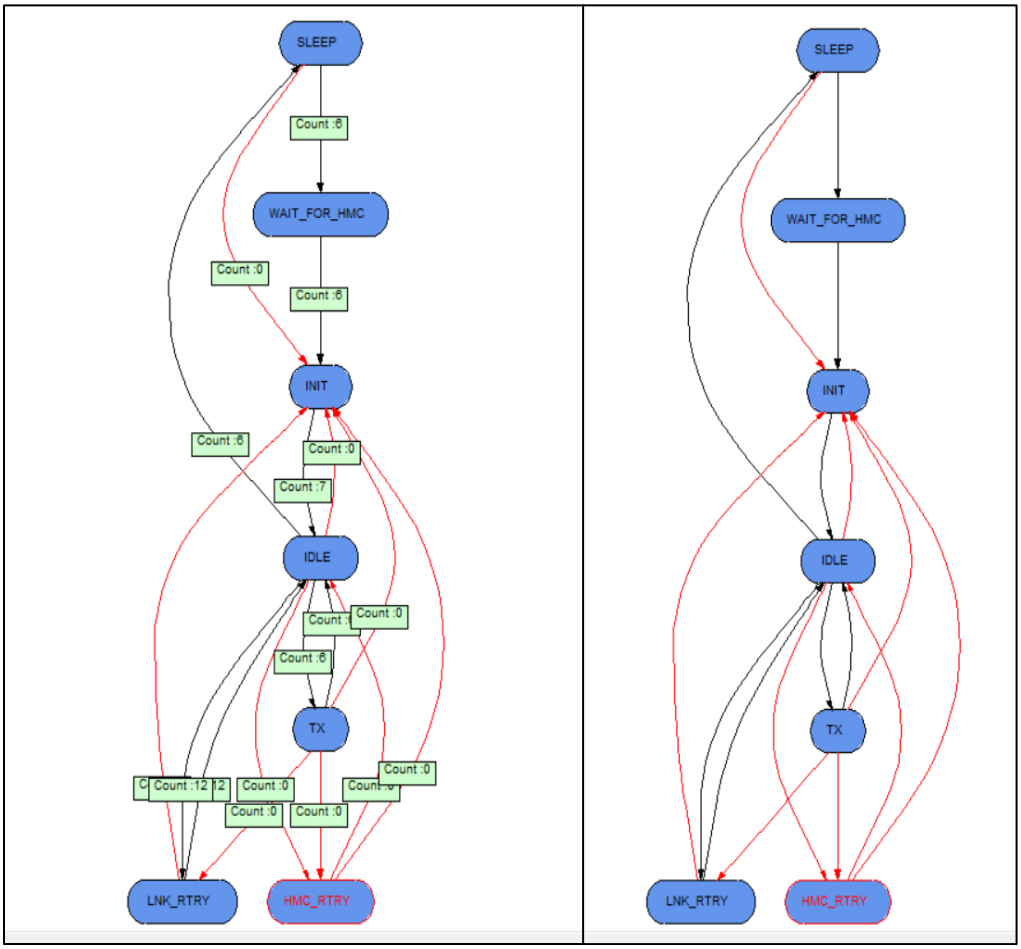


Figure 11.28: state FSM coverage from simple test

To explain the figure, the black arrows and counts means this transition is covered and the red arrows and zero counts means it is not covered. If you notice most of the red arrows are headed to one state which is the INIT state. We figured out that this is the reset state and the reason of this transition would be timing the reset during all the other states to reach INIT so for that we had to do directed tests to hit it later.

For the 2 initialization FSMs it is the same way for the reset state. Shown in (Figure 11.29). All the logical states and transitions are covered except the reset transitions.

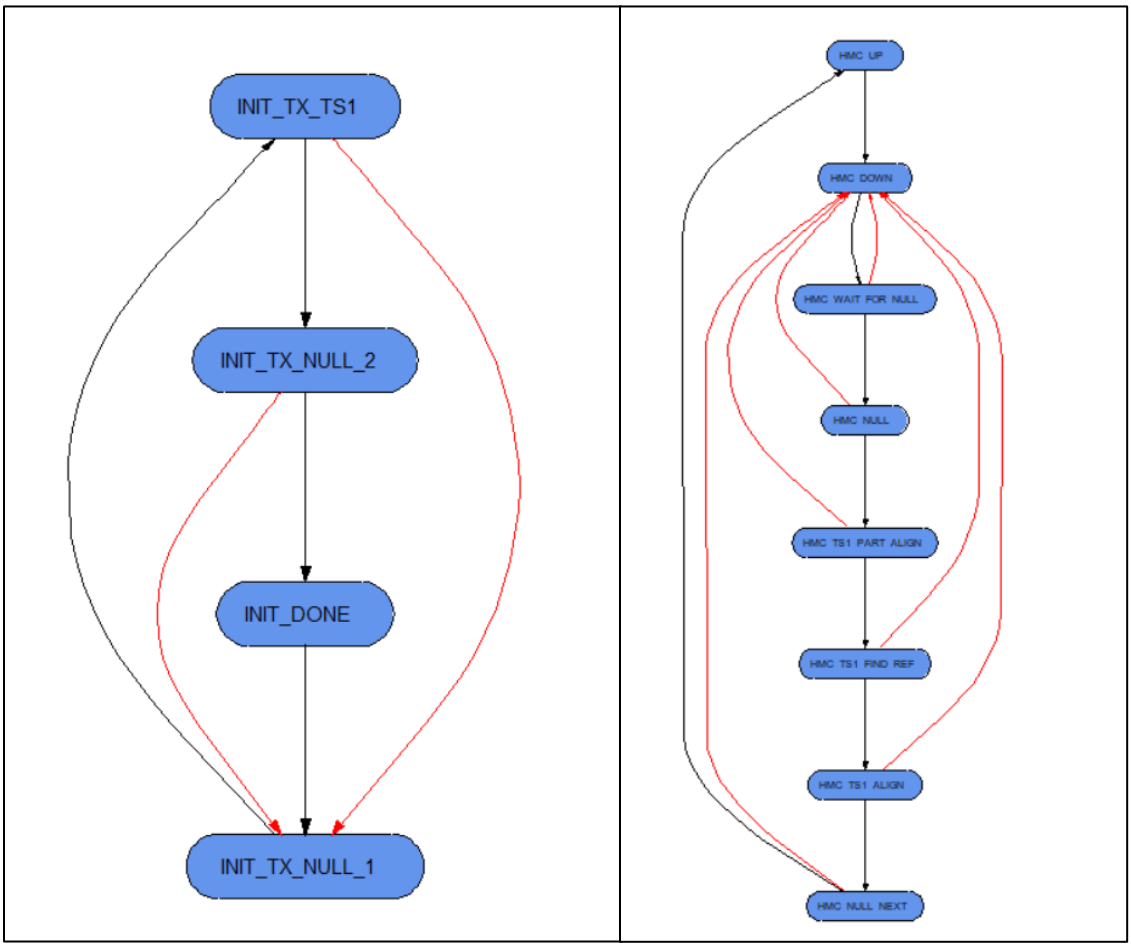


Figure 11.29: init state machines after simple test

11.3.4 Add tests with more constraints

We first focused to make the functional coverage a litter higher. By making some new tests and constraints methods in the sequences to hit the missing commands some examples will be shown here. The base axi sequence constraint part is shown in (Figure 11.30) and in (Figure 11.31).

```
10  typedef enum {
11          WRITE,
12          POSTED,
13          MISC,
14          READ,
15          MODE_READ,
16          ALL_TYPES
17  } request_class_e;
18
19  rand request_class_e req_class;
20
21  constraint req_class_c {
22      req_class == ALL_TYPES;
23  }
```

Figure 11.30: variable that gets randomized in axi sequence

```
axi_base_seq.svh
46  // cerate request in the queue hmc_items
47  function void starting ();
48     hmc_items = new[num_packets];
49     foreach (hmc_items[i]) begin
50     hmc_items[i] = hmc_pkt_item_request::type_id::create($psprintf("hmc_item[%0d]", i));
51     if (req_class == ALL_TYPES) begin
52        assert(hmc_items[i].randomize());
53     end else begin
54        assert(hmc_items[i].randomize() with {
55                                    if (req_class == WRITE) {command inside {WR16, WR32, WR48, WR64, WR80, WR96, WR112, WR128};}
56                                    if (req_class == POSTED) {command inside {P_WR16, P_WR32, P_WR48, P_WR64, P_WR80, P_WR96, P_WR112, P_WR128};}
57                                    if (req_class == MISC) {command inside {MD_WR, BWR, DUAL_2ADD8 , SINGLE_ADD16, P_BWR, P_DUAL_2ADD8, P_SINGLE_ADD16};}
58                                    if (req_class == READ) {command inside {RD16, RD32, RD48, RD64, RD80, RD96, RD112, RD128};}
59                                    if (req_class == MODE_READ) {command inside {MD_RD};}
60                                 });
61     end
62     //`uvm_info(get_type_name(),$psprintf("%0d HMC packets command", hmc_items[i].command), UVM_MEDIUM)
63     end
64  endfunction : starting
```

Figure 11.31: show how randomization inside axi seq is based on the randomized variable

With this if we just extended this sequence and wrote a new constraint for this variable, we can make the randomization focus on certain commands and make the probability higher to achieve them such as in (Figure 11.32).

And what is left is to run a similar test to the simple test but use this sequence instead.

axi_read_seq.svh

```
class axi_read_seq extends  axi_base_seq;

  `uvm_object_utils(axi_read_seq)


  constraint req_class_c {
    req_class inside {READ, MODE_READ};
  }

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


 endclass : axi_read_seq
```

Figure 11.32: extending from axi base sequence and re write constraints

11.3.5 Reset Tests

As mentioned before some sequences were made to directly hit some of the reset states in the FSMs one example can be shown here is covering the reset from sleep state to init state directly without going into the wait_for_hmc state first.

base_seq.svh

```
class base_seq extends  uvm_sequence #(uvm_sequence_item);

  `uvm_object_utils(base_seq)

  rf_reg_block rf_rb;

  env_cfg m_cfg;

  virtual system_if sys_if;

  // Properties used by the various register access methods:
  rand uvm_reg_data_t data;  // For passing data
  uvm_status_e status;       // Returning access status

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

  task body();
    if(m_cfg == null) begin
      `uvm_fatal(get_full_name(), "env_config is null")
    end
    rf_rb = m_cfg.rf_rb;

  endtask : body

  task activate_reset(string parent);
    `uvm_info(parent, "ENTER RESET MODE", UVM_MEDIUM)
      sys_if.res_n  <= 1'b0;
      #500ns;
      @(posedge sys_if.clk)
      sys_if.res_n <= 1'b1;
    `uvm_info(parent, "EXIT RESET MODE", UVM_MEDIUM)
  endtask : activate_reset

endclass : base_seq
```

Figure 11.33: activate reset in base sequence

This is done by basically copying the content of the sleep sequence of the register file but instead of exiting sleep by lowering the set_hmc_sleep field inside the control register, activate the reset of the whole system and this way we start from the beginning.

We implemented a task in the base classes to activate the reset to call it directly from sequences and it can be shown in (Figure 11.33).

11.4 Results

11.4.1 Features and Configurations

Here we will present all the final summary of code, functional as well as assertions coverage. But first let's show the features of the dut that we intended to test in (Table 11.1)

Table 11.1: covered features in tests

Features	Tested
Full link-training, sleep mode and link retraining	✓
16Byte up to 128Byte read and write (posted and non-posted) transactions	✓
Posted and non-posted bit-write and atomic requests	✓
Mode read and write	✓
Error response	✗
Full packet flow control	✓
Packet integrity checks (sequence number, packet length, CRC)	✓
Fully automatic error handling and link retry	○
Control and detect lane reversal and lane polarity	✗

The green check mark means it is fully covered, the red cross means it is not tested and the yellow circle means only partially tested.

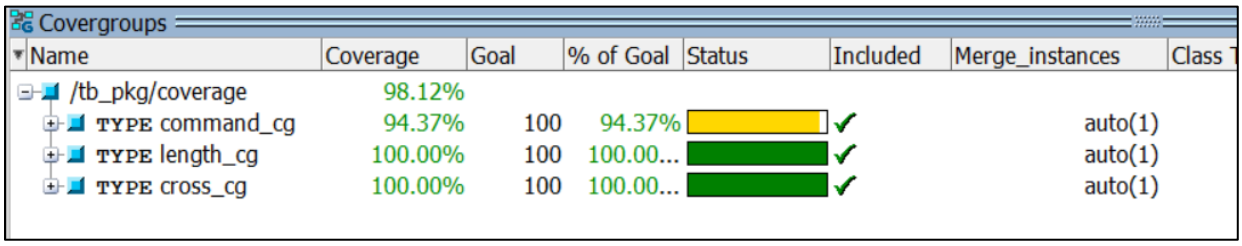
While for the configurations of the DUT is shown in (Table 11.2).

Table 11.2: openhmc tested configurations

AXI Interface Configurations	Tested
2 FLITs per Word / 256-bit Datapath	✗
4 FLITs per Word / 512-bit Datapath	✓
6 FLITs per Word / 768-bit Datapath	✗
8 FLITs per Word / 1024-bit Datapath	✗

11.4.2 Functional Coverage

Shown in (Figure 11.34). The summary of functional coverage for the 3 covergroups.



Covergroups

Name	Coverage	Goal	% of Goal	Status	Included	Merge_instances	Class T
/tb_pkg/coverage	98.12%						
TYPE command_cg	94.37%	100	94.37%		✓	auto(1)	
TYPE length_cg	100.00%	100	100.00...		✓	auto(1)	
TYPE cross_cg	100.00%	100	100.00...		✓	auto(1)	

Figure 11.34: summary of the three covergroups

The missing commands are shown in (Figure 11.35). As mentioned before, we didn't intend to test the error response packet and for the PRET flow packet type we did implement it to be sent but our codes might not be fully debugged and working properly and due to the lack of time we couldn't trace the issue.

Covergroups

Name	Coverage	Goal	% of Goal	Status	Included
/tb_pkg/coverage	98.12%				
TYPE command_cg	94.37%	100	94.37%		✓
CVP command_cg::Flow	75.00%	100	75.00%		✓
bin flow[NULL]	2758	1	100.00...		✓
bin flow[PRET]	0	1	0.00%		✓
bin flow[TRET]	203	1	100.00...		✓
bin flow[IRTRY]	1536	1	100.00...		✓
CVP command_cg::Write_Requests	100.00%	100	100.00...		✓
CVP command_cg::Misc_Write_Requests	100.00%	100	100.00...		✓
CVP command_cg::Posted_Write_Requests	100.00%	100	100.00...		✓
CVP command_cg::Posted_Misc_Write_Req...	100.00%	100	100.00...		✓
CVP command_cg::Mode_Read_Request	100.00%	100	100.00...		✓
CVP command_cg::Read_Requests	100.00%	100	100.00...		✓
CVP command_cg::Response_Commands	80.00%	100	80.00%		✓
bin Response_Commands[RD_RS]	27	1	100.00...		✓
bin Response_Commands[WR_RS]	42	1	100.00...		✓
bin Response_Commands[MD_RD_RS]	11	1	100.00...		✓
bin Response_Commands[MD_WR_RS]	5	1	100.00...		✓
bin Response_Commands[ERROR_RS]	0	1	0.00%		✓
TYPE length_cg	100.00%	100	100.00...		✓
TYPE cross_cg	100.00%	100	100.00...		✓

Figure 11.35: messing coverage commands

11.4.3 Code Coverage

After adding the reset tests and covering most of the finite state machine the code coverage jumped and total summary is shown in (Figure 11.36).

```
TOTAL COVERGROUP COVERAGE: 98.12%  COVERGROUP TYPES: 3

TOTAL ASSERTION COVERAGE: 100.00%  ASSERTIONS: 41

Total Coverage By Instance (filtered view): 87.14%
```

Figure 11.36: final code coverage summary

It is hard to find and sort by the top module and show the code coverage of the most important modules so we will try to find them and summarize from top to bottom starting from (Figure 11.37).

```
================================================================================
=== Instance: /tb_top/dut
=== Design Unit: work.openhmc_top
================================================================================
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Expressions                      5         5         0   100.00%
    Statements                       1         1         0   100.00%
```

Figure 11.37: code coverage of openHMC top

Then the register file module coverage is shown in (Figure 11.38).

```
================================================================================
=== Instance: /tb_top/dut/openhmc_rf_I
=== Design Unit: work.openhmc_rf
================================================================================
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Branches                        40        39         1    97.50%
    Conditions                      20        20         0   100.00%
    Expressions                     30        26         4    86.66%
    Statements                     131       128         3    97.70%
```

Figure 11.38: code coverage of openHMC rf

The TX and RX links are shown in (Figure 11.39).

```
================================================================================
=== Instance: /tb_top/dut/rx_link_I
=== Design Unit: work.rx_link
================================================================================
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Branches                      2205      1159      1046    52.56%
    Conditions                      37        19        18    51.35%
    Expressions                   3121      1068      2053    34.21%
    FSM States                       8         8         0   100.00%
    FSM Transitions                 14        14         0   100.00%
    Statements                    1433      1402        31    97.83%
================================================================================
=== Instance: /tb_top/dut/tx_link_I
=== Design Unit: work.tx_link
================================================================================
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Branches                       201       179        22    89.05%
    Conditions                      79        56        23    70.88%
    Expressions                     41         3        38     7.31%
    FSM States                      11        11         0   100.00%
    FSM Transitions                 23        19         4    82.60%
    Statements                     343       311        32    90.67%
```

Figure 11.39: rx and tx links code coverage

11.4.4 Finite State Machine Coverage

Will present here the snapshots of the three FSMs of the DUT after finishing and merging all the tests.

First the state FSM can be seen in (Figure 11.40).

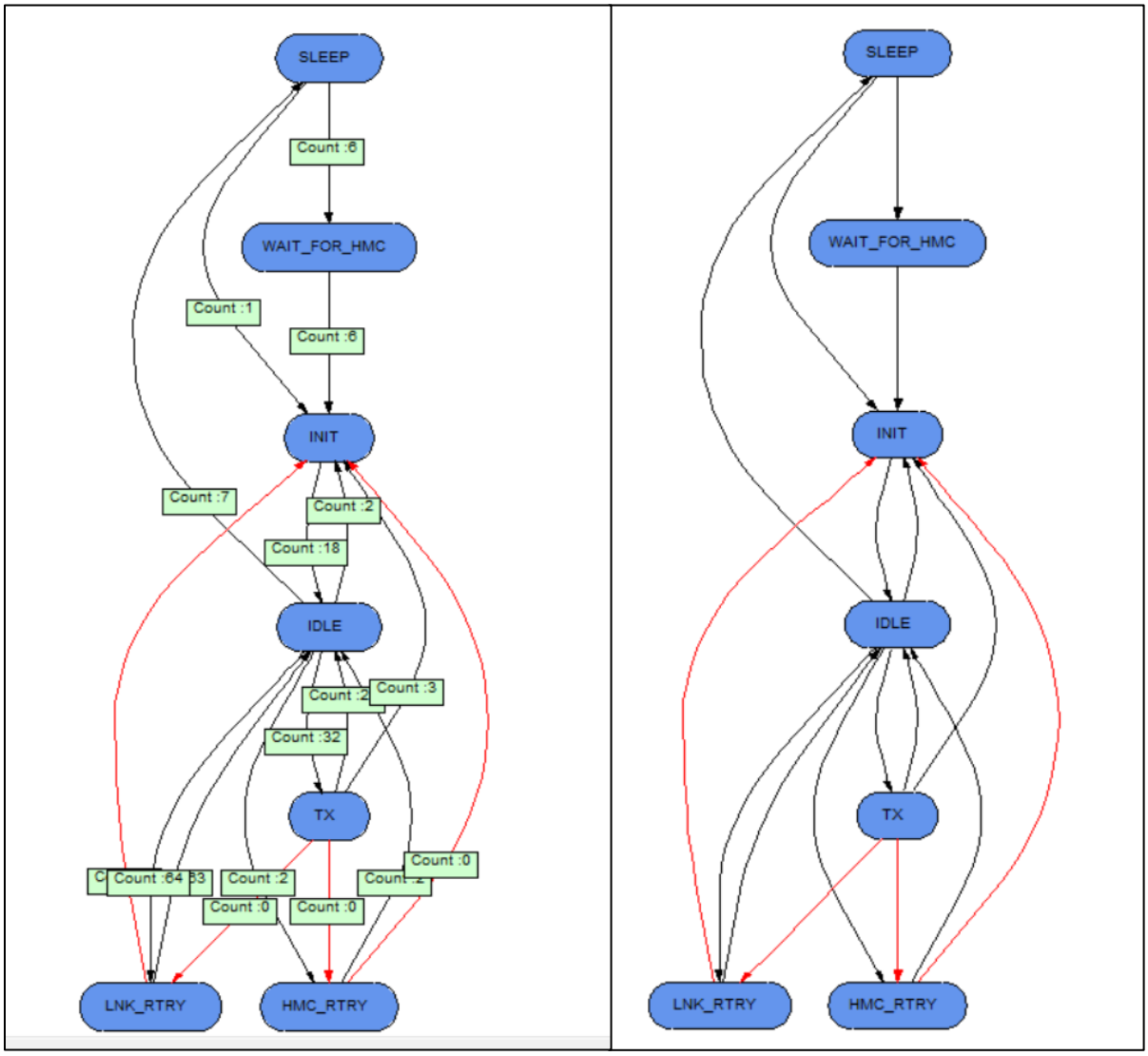


Figure 11.40: final state FSM coverage

The only untested transitions in the state FSM two of them are because of reset and the dut really didn't make any signal that we can see to know the exact timing to enter the reset. As all the others it is easy to hit because inside the registers of the dut we can hit the timing when it exactly enter any state in the initialization from reading the status init register and we can easily time the idle and tx states because we know when we send the requests packets from axi. Finally, it is also hard to time moving from tx into the link retry state and will need a specific scenario which wasn't tried yet.

Next is both the initialization FSMs which are fully covered shown in (Figure 11.41).

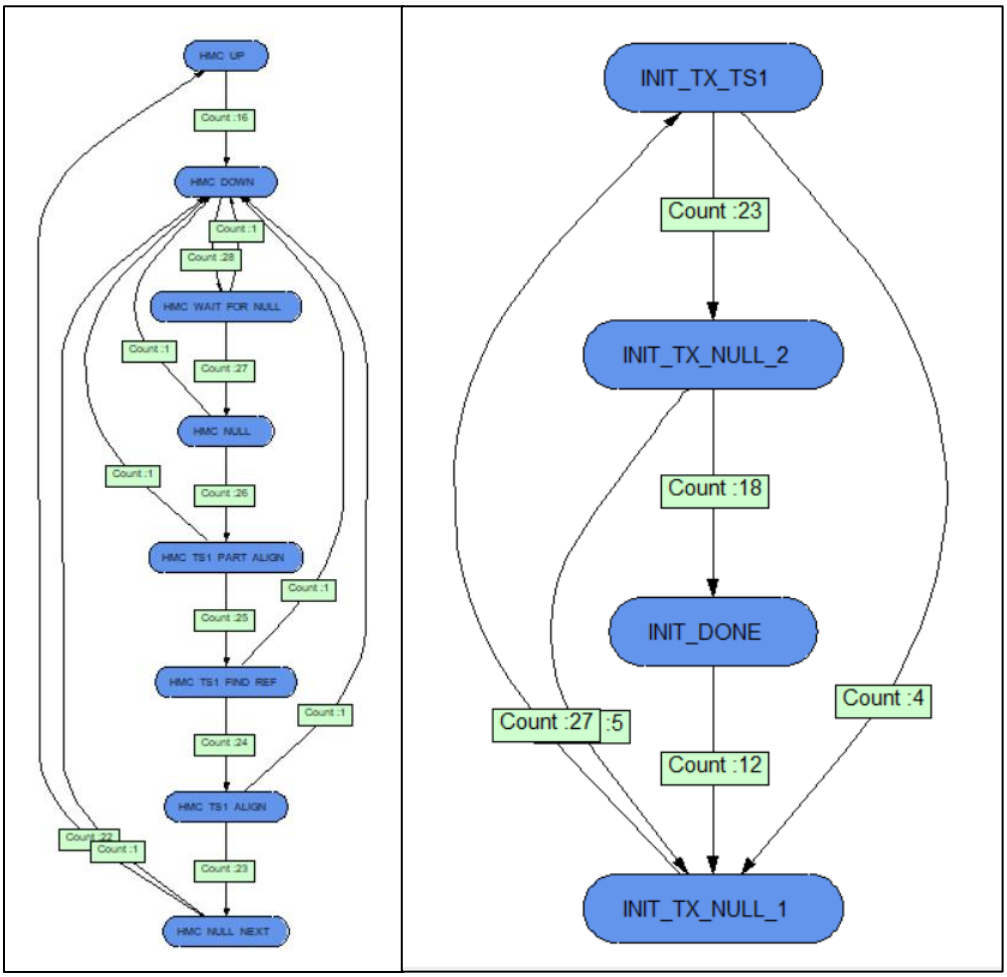


Figure 11.41: initialization states fully covered

11.4.5 Assertions

In total we wrote 41 assertions all of them passed successfully and 10 of them are concurrent written in a single file and used the bind method mentioned in [7]. For the immediate assertions they are like if conditions that produce an error message when they fail. Here is a snapshot from the assertions tab and only the concurrent assertions in questasim in (Figure 11.42).

Assertions

Name	Assertion Type	Language	Failure	Pass Count	Assertion Expression	Enable
/tb_top/dut/openhmc_sva_1/no_simultaneous_read_and_write	Concurrent	SVA	0	105912	assert(@(posedge clk_...	on
/tb_top/dut/openhmc_sva_1/access_complete_one_cycle_after_operation	Concurrent	SVA	0	4924	assert(@(posedge clk_...	on
/tb_top/dut/openhmc_sva_1/invalid_address_assert_if_wrong_write_operation	Concurrent	SVA	0	11	assert(@(posedge clk_...	on
/tb_top/dut/openhmc_sva_1/tx_data_hold_until_ready_active	Concurrent	SVA	0	42	assert(@(posedge clk_...	on
/tb_top/dut/openhmc_sva_1/tx_user_hold_until_ready_active	Concurrent	SVA	0	42	assert(@(posedge clk_...	on
/tb_top/dut/openhmc_sva_1/tx_valid_hold_until_ready_active	Concurrent	SVA	0	42	assert(@(posedge clk_...	on
/tb_top/dut/openhmc_sva_1/rx_valid_hold_until_ready_active	Concurrent	SVA	0	70	assert(@(posedge clk_...	on
/tb_top/dut/openhmc_sva_1/rx_user_hold_until_ready_active	Concurrent	SVA	0	70	assert(@(posedge clk_...	on
/tb_top/dut/openhmc_sva_1/rx_data_hold_until_ready_active	Concurrent	SVA	0	70	assert(@(posedge clk_...	on
/tb_top/dut/openhmc_sva_1/invalid_address_assert_if_wrong_read_operation	Concurrent	SVA	0	1	assert(@(posedge clk_...	on

Figure 11.42: all concurrent assertions results

Chapter 12
Conclusion

In this final chapter, we will discuss what we learned during this project and some of the problems we faced and finally what we could've done better and some future work and advices.

12.1 Points Learned

While we started this project having zero knowledge and experience about the functional verification and UVM field. With the help of our mentors, we progressed quickly and step by step to start with the basics skills and then we could have enough knowledge to be able to perform self-learning by reading more papers, references, guides and websites about the verification topic to achieve all the results which is considered a fair job for a start. In summary, all the skills we learned

- Learned about functional verification, its importance and role in the IC field.
- Learned the SystemVerilog language that is far more complex and contains many features that the older Verilog.
- Learned UVM and built first a simple environment of an ALU before starting on our graduation project.
- Faced multiple compilation and simulation errors that we learned to search online for solutions and debug them throughout the project.
- Learned the skill of self-learning that we don't specifically need an instructor to teach us a topic and can search on our own.
- Learned to read a user manual of a simulation tool (Questasim) and write its scripts.

12.2 Problems Faced

Throughout the project and dealing with this specific design we faced multiple problems related to the design that will be summarized here

- Bad documentation and explanation in some aspects of the OpenHMC design as the whole design was explained in about 16 papers only which don't match the amount of code written and needed to explain more as some aspects like the transceiver wasn't mentioned in the specs and it was left out for us to figure out how it works.
- An advice for graduation projects was that always because the specs is hard to understand is to search for more online videos, papers and resources that explain it, the problem was the lack of this kind of resources and the only source was the specs document.

- Hard to time transition between states as we must read from registers which takes time then this output from registers must reach driver somehow, which could've been designed in a better way.
- For the transceiver, there wasn't a direct way to connect to the HMC as mentioned in the specs. As in the specs, the HMC pins consists of only the lanes and it transfer data in a serialized way. So, to connect the HMC with the DUT we had to write a serializer and de-serializer modules to close this gap.

Next problem is related to the company when it provided us its version of the simulation tools

- The problem was to access the tool we had to go through many steps each time to only open the virtual machine, then after unlocking everything the ability of moving files from our devices to the virtual machine was closed and the ability to copy and paste text was also closed. So, to use this tool we must create new files inside the virtual machine and manually type all the RTL codes and our Testbench codes which isn't logical given that these codes consist of tens of thousands of lines. In addition, because we signed an NDA not to share any screenshots from the virtual machine even if we used it we can't report our results.

12.3 Advices and Future Work

In this section, we will write some notes about things we could've done throughout the project and if given more time we could do

- During compilation and simulation, we faced many errors. What we should have done is each time we face an error we document it and document the solution along with it and the duration it took us to solve this error.
- There are many aspects of functional verification and coding methods we didn't try during this project during the lack of time. One of them is to trace the rtl code and manually read each statement, branch or condition that we didn't hit and try to test it with a scenario.
- We took long time during the verification plan period because of the lack of examples as we didn't know exactly how a plan should look like. So, for that we needed to ask for more examples and check more already written plans.
- A term in verification is called error injection. Which is manually injecting errors in our stimulus and check how the DUT reacts. It is a common way in verification and we could've tried it if given more time.
- Of course, adding more tests and covergroups as the current ones are very simple and basic.

References

[1] C. Spear and G. Tumbush, "SystemVerilog for Verification Third Edition," 2012. [Online]. Available: https://drive.google.com/file/d/1cwdfSyWnGsfXaZ3GjqDNjBQ_WfLTiqgz/view [Accessed July 2023].

[2] ChipVerify, "SystemVerilog Tutorial," [Online]. Available: https://www.chipverify.com/systemverilog/systemverilog-tutorial. [Accessed July 2023].

[3] ChipVerify, "UVM Tutorial for Beginners," [Online]. Available: https://www.chipverify.com/uvm/uvm-tutorial. [Accessed July 2023].

[4] Computer Architecture Group, "OpenHMC," github, [Online]. Available: https://github.com/unihd-cag/openhmc.git. [Accessed July 2023].

[5] The Hybrid Memory Cube Consortium (HMCC), "Hybrid Memory Cube Specification 1.0," [Online]. Available: https://web.archive.org/web/20130513053443/http://www.hybridmemorycube.org/files/SiteDownloads/HMC_Specification%201_0.pdf. [Accessed July 2023].

[6] Verification Academy, "Registers," Siemens, [Online]. Available: https://verificationacademy.com/cookbook/registers. [Accessed July 2023].

[7] C. E. Cummings, "Sunburst-design SVA Bind," 2009. [Online]. Available: http://www.sunburst-design.com/papers/CummingsSNUG2009SJ_SVA_Bind.pdf. [Accessed July 2023].

[8] GitHub, "lowRISC SystemVerilog Coding Style Guide for Design Verification," [Online]. Available: https://github.com/lowRISC/style-guides.git. [Accessed July 2023].

[9] GitHub, "Graduation Project GitHub Repo," [Online]. Available: https://github.com/alaarabie/Graduation_project.git. [Accessed July 2023].

[10] GitHub, "AlDanial/cloc - GitHub," [Online]. Available: https://github.com/AlDanial/cloc.git [Accessed July 2023].

[11] C. E. Cummings, "Sunburst-design Using UVM Virtual Sequencers & Virtual Sequences," 2016. [Online]. Available: http://www.sunburst-design.com/papers/CummingsDVCon2016_Vsequencers.pdf. [Accessed July 2023].

[12] Verification Academy, "Handling Parameterisation," Siemens, [Online]. Available: https://verificationacademy.com/cookbook/handlingparameterisation. [Accessed July 2023].

[13] Sublime Text, "Sublime Text - Text Editing, Done Right," [Online]. Available: https://www.sublimetext.com/. [Accessed July 2023].

[14] SCRIBD, "Questa Sim Ref | PDF | License | Proprietary Software," [Online]. Available: https://www.scribd.com/document/517338649/Questa-Sim-Ref#. [Accessed July 2023].

[15] TCL Developer Xchange, "Tcl Developer Site," [Online]. Available: https://www.tcl.tk/. [Accessed July 2023].

[16] R. Salemi, "The UVM Primer," 2013. [Online]. Available: https://drive.google.com/file/d/12OFuTj9VnTaohqk32bfBPUV3Bworvs5t/view. [Accessed July 2023].