Verification of Hybrid Memory Cube Controller using Universal Verification Methodology

Team Members:

Islam Ahmed Elfeky
Kholoud Ebrahim Darwesh
Mohamed Ibrahim Mohamed
Omnia Mohamed Selim
Omnia Salah Ibrahim
Rana Wagdy Shafik
Tarek Emad Mohamed
Youmna Mohamed Refaat

Supervised by: Dr. Mohamed El-Banna

Sponsor: Si-vision

DECLARATION

We hereby certify that this material, which we now submit for assessment on the program of study leading to the award of Bachelor of Science in Electronics and Communication Engineering At Alexandria university is entirely our own work, that we have exercised reasonable care to ensure that the work is original, and does not to the best of our knowledge breach any law of copyright, and have not been taken from the work of others and to the extent that such work has been cited and acknowledged within the text of our work.

Signed: by all students

Islam Ahmed Elfeky	
Kholoud Ebrahim Darwesh	
Mohamed Ibrahem Mohamed	
Omnia Mohamed Selim	
Omnia Salah Ibrahim	
Rana Wagdy Shafik	
Tarek Emad Mohamed	
Youmna Mohamed Refaat	

Date: 9, July 2023.

ACKNOWLEDGMENT

The success and outcome of this project required a lot of guidance and assistance from many people, and we are extremely privileged to have got this all through the completion of our project. Alee that we have done is only due to such supervision and assistance and we should not forget to thank them.

We would like to express our very great appreciation and deep gratitude to **Eng. Ahmed Abdelrahman, Si-Vision, Eng. Ahmed Abdelbaky, Si-Vision, Eng. Hagar Fathy, Si-Vision, Eng. Mohamed Younis, Si-Vision, Eng. Yahia Zakaria , Si-Vision, Eng. Mostafa Sarhan, Si-Vision, and Eng. Hatem Ali, Si-Vision** for their patient guidance, enthusiastic encouragement, and useful critiques of this project work.

We are thankful to **Prof. Dr.Mohamed Ismail El-Banna**, Department Electronics and Communication Engineering, Alexandria University, for supervising our project and providing us with the opportunity to learn something new.

Finally, we are grateful for all the encouragement and support we got whether from our families, teaching staff and our colleagues.

ABSTRACT

Digital verification consists of acquiring a reasonable confidence that a circuit will function correctly, under the assumption that no manufacturing fault is present. Verification plays a vital role as most of the time is engaged in verification. So, ASIC or SOC verification companies adopted several methodologies for re-usability purposes. The latest methodology adopted is UVM due to its advantages. In this book, we propose our verification plan along with the SystemVerilog & UVM verification environments architectures for Open-HMC Memory controller. These verification environments use coverage driven random verification to verify the functionality of the Memory controller, in translating the requests coming from the host using AXI protocol and send these requests to the HMC memory cube and finally translating the responses from the HMC memory cube to the host by using the AXI protocol, use assertion-based verification to check timing constraints between the Memory Controller and HMC Memory Cube (which in our case is replaced with reactive slave agent), Also, in this book you can find the detailed implementation for each environment and the simulation results for testcases.

Table of Contents

List of Figures

List of Tables

LIST OF ACRONYMS

FLIT	flow control unit or flow control digit
BFM	Bus Functional Model
CAG	Computer Architecture Group
CDR	Clock-Data Recovery
DEMUX	De-Multiplexer
DUT	Device Under Test
FPW	FLITs Per Word
FRP	Forward Retry Pointer
FSM	Finite State Machine
HMC	Hybrid Memory Cube
HMCC	Hybrid Memory Cube Consortium
LFSR	Linear Feedback Shift Register
LUT	Look-Up Table
PLL	Phase-Locked Loop
RF	Register File
RRP	Return Retry Pointer
RTC	Return Token Count
RX	Receive
SEQ	Sequence Number
TRET	Token Return
TX	Transmit
UVM	Universal Verification Methodology
TS1	Training sequence 1
TRET	Token Return
CRC	cyclic redundancy check
SLID	source link identifier
FIFO	First In Frist Out
SOC	System On Chip

Chapter 1: Overview of Digital Verification

1.1 Introduction

Verification is a very crucial step in the digital design flow of any digital circuit. It is the formal process of evaluating and validating a hardware design to ensure that it meets its specified requirements and specifications. It involves creating a testbench or test environment that simulates the behavior of the hardware design under different conditions and test cases. The testbench generates input stimuli that are applied to the design, and the output response of the design is compared to the expected results. The goal of verification is to identify and correct any design issues or errors before the product is released to the market. Verification can be performed at different levels of abstraction in the design hierarchy and can involve different types of tests, such as functional tests, performance tests, and reliability tests. Also, there are several verification techniques used in the industry to validate the correctness of RTL designs. Each technique has its strengths and weaknesses, and choosing the right method for your project depends on factors such as the design complexity, required confidence level, and available resources. So, what are those techniques?

1.2 Verification Techniques

Here are some commonly used verification techniques in digital IC design:

- **Simulation:** Simulation is the most common RTL verification technique, involving the execution of testbenches that apply a set of test vectors to the RTL design and compare the resulting outputs against the expected values. Simulation can be done at various abstraction levels, including behavioral, RTL, and gate-level. While simulation is relatively straightforward and can catch many design errors, it may not provide full coverage of all possible input combinations and corner cases.
- **Formal Verification:** Formal verification is a mathematical approach to proving the correctness of a design by exhaustively exploring all possible input combinations and states. This technique uses formal methods, such as theorem proving and model checking, to ensure that the design meets its specifications. While formal verification can provide a higher level of confidence in design correctness, it can be more complex and time-consuming than simulation, especially for large and complex designs.
- **Emulation:** Emulation involves using specialized hardware platforms, such as FPGA-based emulators or custom-built emulators, to run the RTL design at near real-time speeds. This

allows for more extensive testing and faster verification of complex designs. However, emulation can be expensive and may require additional expertise to set up and manage the emulation environment. Now, after this overview about the Verification we will explain the Design Flow briefly to know at which step is the Verification in the Design Flow.

1.3 Design Flow

The digital design flow is a flow for designing digital circuits and systems, from specification to final implementation. It typically involves the following main steps shown in Figure 1.1.

Design specifications: The design process begins with the design specifications, which specify the requirements and functionality of the digital circuit or system. This involves defining the inputs, outputs, and behavior of the system.

Functional design: The functional design phase involves creating a high-level conceptual design of the system. This may involve using block diagrams, state diagrams, or other modeling techniques to create a functional specification of the system.

RTL Design: The RTL (Register Transfer Level) design phase involves creating a detailed digital design of the system. During this phase, the architectural description is further refined: memory elements and functional components of each model are designed using a Hardware Description Language (HDL). This phase also sees the development of the clocking system of the design and architectural trade-offs such as speed/power.

RTL verification: The RTL design is then verified using simulation and other verification techniques to ensure that it behaves as expected and meets the design specifications. This phase consists of acquiring a reasonable confidence that a circuit will function correctly, under the assumption that no manufacturing fault is present. The underlying motivation is to remove all possible design errors before proceeding to expensive chip manufacturing.

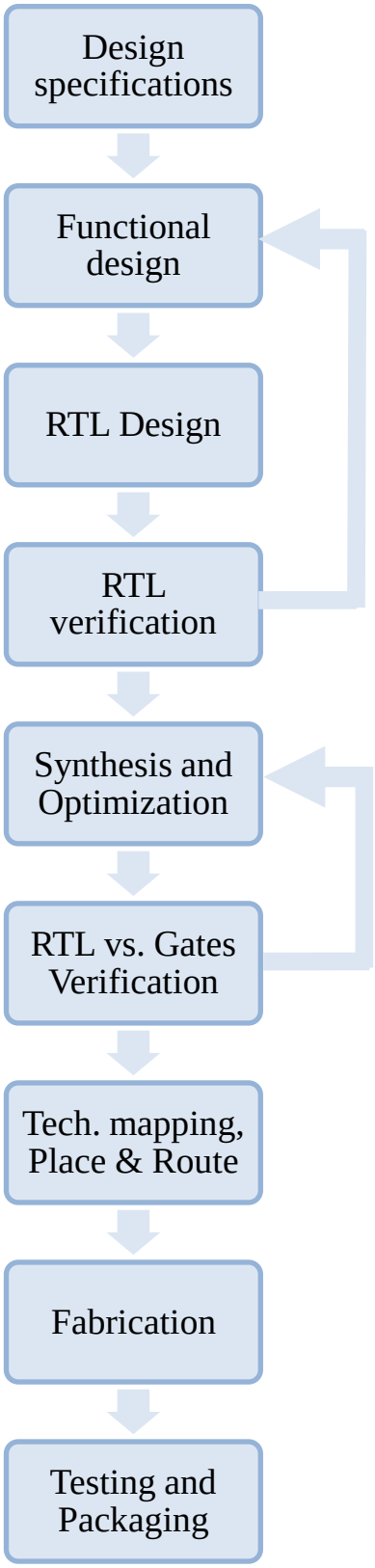


Figure 1. 1: Design Flow

Each time functional errors are found the model needs to be modified to reflect the proper behavior. During RTL verification, the verification team develops various techniques and numerous suites of tests to check that the design behavior corresponds to the initial specifications. When that is not the case, the functional design model must be modified to provide the correct behavior specified and the RTL design updated consequently. It is also possible that the RTL verification phase reveals incongruences or overlooked aspects in the original set of specifications and this latter one needs to be updated instead. Our project focuses on this phase from the design flow.

Synthesis and Optimization: Once the RTL design is verified, it is synthesized using a synthesis tool to generate a gate-level netlist. The netlist represents the circuit as a collection of logic gates and flip-flops. The netlist is then optimized using a logic optimization tool to reduce the size and power consumption of the circuit.

RTL vs. Gates Verification: Then the synthesized model needs to be verified. The objective of RTL versus gates verification, or equivalency checking is to guarantee that no errors have been introduced during the synthesis phase. It is an automatic activity requiring minimal human interaction that compares the pre-synthesis RTL description to the post-synthesis gate level description in order to guarantee the functional equivalence of the two models.

Tech. mapping, Place & Route: The gate-level netlist is then mapped to a specific technology library and placed and routed on a chip using a layout tool. This involves placing the gates and routing the interconnections between them.

Fabrication: The chip is fabricated using a semiconductor fabrication process once the layout is complete. This involves creating multiple layers of metal interconnects and doping the silicon to create the necessary transistors and other components.

Testing and Packaging: Once the chip is fabricated, it is tested to ensure that it meets the specifications and requirements. The chip is then packaged and ready for use in a digital system.

Overall, the digital design flow is a complex and iterative process that involves multiple tools and techniques to design, optimize, verify, fabricate, test, and package a digital circuit or system.

1.4 History of Languages Used in Verification

The history of languages used in verification can be traced back to the early days of digital electronics. In the 1960s and 1970s, hardware design was typically done using low-level languages such as assembly language and machine code. Verification was largely a manual process, involving the use of oscilloscopes and other test equipment to observe the behavior of the hardware.

In the 1980s, the emergence of high-level hardware description languages (HDLs) such as VHDL (VHSIC Hardware Description Language) and Verilog changed the landscape of hardware design and verification. These languages provided a way to describe the behavior of digital circuits and systems at a higher level of abstraction, making it easier to design and verify complex systems.

Initially, Verilog was the predominant language used for IC verification. It was developed in the mid-1980s by Phil Moorby and Prabhu Goel and provided a way to model the behavior of digital circuits and systems using procedural programming constructs. Verilog quickly gained popularity in the IC verification community and became the de facto standard language for IC verification.

In the early 2000s, SystemVerilog was introduced as an extension of Verilog to address the limitations of the language in modeling complex behaviors, such as concurrency and data types. SystemVerilog incorporated new features such as object-oriented programming constructs, constrained random testing, and assertions, which made it easier to model complex designs and perform more thorough verification. As a result, SystemVerilog quickly gained popularity and became the new standard language for IC verification.

More recently, as the complexity of designs has continued to increase, there has been a growing need for more efficient and scalable verification methodologies. This has led to the development of the Universal Verification Methodology (UVM), which is a standardized methodology for IC verification that provides a structured approach to developing and managing verification environments. UVM is built on top of SystemVerilog and provides a set of reusable components and guidelines for creating efficient and scalable verification environments.

The evolution of languages used in verification has been driven by the need for better modeling and verification capabilities, increasing design complexity, and the need for greater productivity in the verification process. Verilog was the starting point, but as designs became more complex, SystemVerilog and eventually UVM provided the necessary features and methodologies to meet the evolving challenges of IC verification.

1.5 Historical Overview of Verification Methodologies

As shown in Figure 1.2, The world of digital verification underwent significant changes in 2000 when Verisity introduced a dedicated verification language called e, and the associated e Reuse Methodology. The eRM enabled verification teams to create test benches that randomized a design's inputs subject to a set of user-specified constraints, largely automating the verification process, especially if the tests were self-checking and coverage metrics helped to gauge verification thoroughness.

Synopsys also developed a similar approach with its dedicated verification language VERA, developing the Reference Verification Methodology (RVM).

When the Accellera standards organization decided to create SystemVerilog, Synopsys adapted the RVM to the new language, resulting in the Verification Methodology Manual (VMM) for SystemVerilog, which was published jointly with ARM.

Mentor Graphics responded with the Advanced Verification Methodology (AVM), which supported both SystemVerilog and SystemC. Cadence initially advocated SystemC for verification but later acquired Verisity and transformed the eRM into the Universal Reuse Methodology (URM), which supported SystemVerilog in addition to e and SystemC.

Industry convergence began in 2008 when Cadence and Mentor collaborated on the Open Verification Methodology (OVM), available as open source with simple licensing. Accellera selected the OVM as the baseline while incorporating key VMM features, resulting in the widely adopted Universal Verification Methodology (UVM). These developments have played an essential role in advancing the field of digital verification and improving the quality and reliability of digital systems.

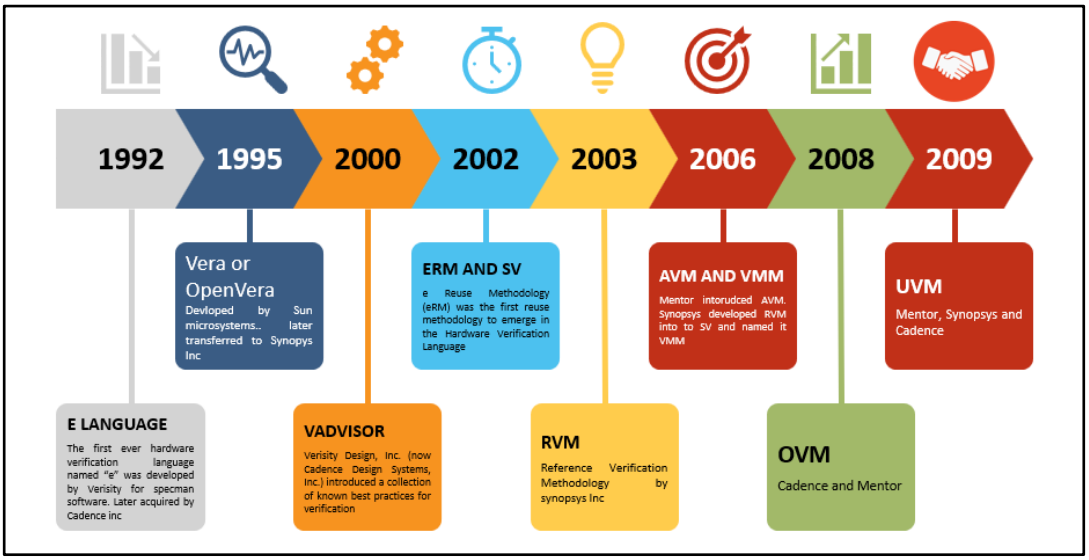


Figure 1. 2: Verification Methodologies History

Chapter 2 : Hybrid Memory Cube

2.1 What is HMC

2.1.1 History

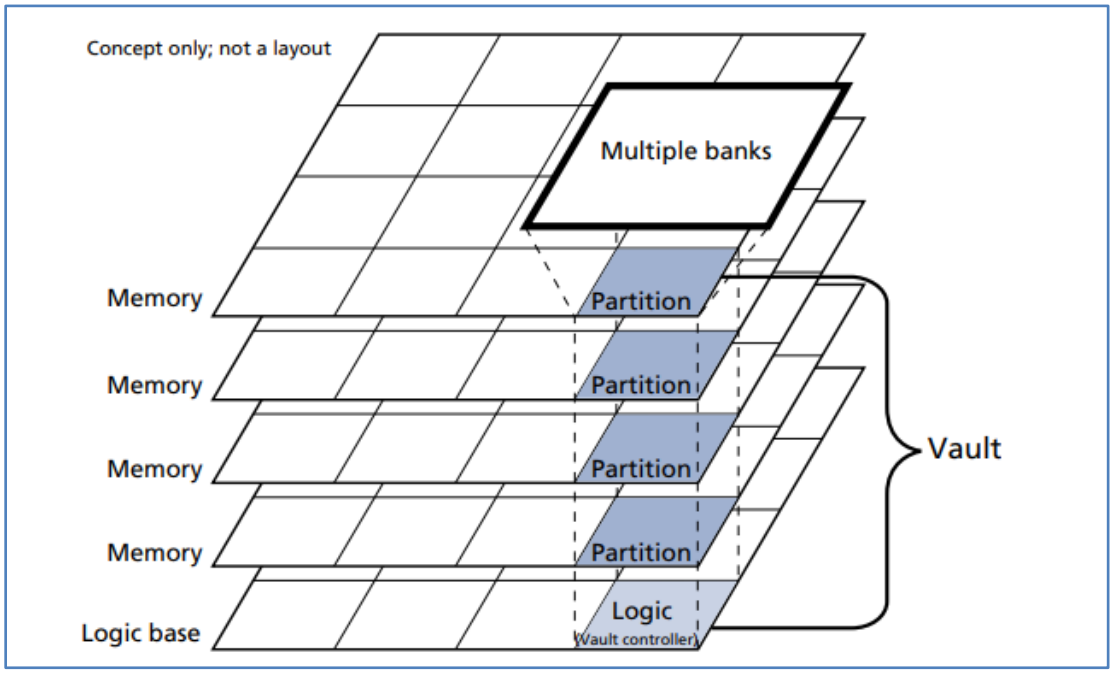


Figure 2. 1: Example HMC Organization

Hybrid Memory Cube (HMC) is a high-performance computer random-access memory (RAM) interface for through-silicon vias (TSV)-based stacked DRAM memory competing with the incompatible rival interface High Bandwidth Memory (HBM).

Hybrid Memory Cube was co-developed by Samsung Electronics and Micron Technology in 2011 and announced by Micron in September 2011. It promised a 15 times speed improvement over DDR3. The Hybrid Memory Cube Consortium (HMCC) is backed by several major technology companies including Samsung, Micron Technology, Open-Silicon, ARM, HP (since withdrawn), Microsoft (since withdrawn), Altera (acquired by Intel in late 2015), and Xilinx. Micron, while continuing to support HMCC, discontinued the HMC product in 2018 when it failed to achieve market adoption.

HMC combines through-silicon vias (TSV) (is a vertical electrical connection (via) that passes completely through a silicon wafer or die. TSVs are high-performance interconnect techniques

used as an alternative to wire-bond and flip chips to create 3D packages and 3D integrated circuits. Compared to alternatives such as package-on-package, the interconnect and device density is substantially higher, and the length of the connections becomes shorter.) and micro bumps to connect multiple (currently 4 to 8) dies of memory cell arrays on top of each other. The memory controller is integrated as a separate die.

HMC uses standard DRAM cells, but it has more data banks than classic DRAM memory of the same size. The HMC interface is incompatible with current DDRn (DDR2 or DDR3) and competing High Bandwidth Memory implementations. HMC technology won the Best New Technology award from The Linley Group (publisher of Microprocessor Report magazine) in 2011.

The first public specification, HMC 1.0, was published in April 2013. According to it, the HMC uses 16-lane or 8-lane (half size) full-duplex differential serial links, with each lane having 10, 12.5 or 15 Gbit/s SerDes. Each HMC package is named a cube, and they can be chained in a network of up to 8 cubes with cube-to-cube links and some cubes using their links as pass-through links. A typical cube package with 4 links has 896 BGA pins and a size of 31×31×3.8 millimeters.

Figure 2. 2: SerDes (Serializer - Deserializer)

The typical raw bandwidth of a single 16-lane link with 10 Gbit/s signaling implies a total bandwidth of all 16 lanes of 40 GB/s (20 GB/s transmit and 20 GB/s receive); cubes with 4 and 8 links are planned, though the HMC 1.0 spec limits link speed to 10 Gbit/s in the 8-link case. Therefore, a 4-link cube can reach 240 GB/s memory bandwidth (120 GB/s each direction using 15 Gbit/s SerDes), while an 8-link cube can reach 320 GB/s bandwidth (160 GB/s each direction using 10 Gbit/s SerDes). Effective memory bandwidth utilization varies from 33% to 50% for smallest packets of 32 bytes; and from 45% to 85% for 128-byte packets.

As reported at the Hot Chips 23 conference in 2011, the first generation of HMC demonstration cubes with four 50 nm DRAM memory dies and one 90 nm logic die with total capacity of 512 MB and size 27×27 mm had power consumption of 11 W and was powered with 1.2 V.
Engineering samples of second generation HMC memory chips were shipped in September 2013 by Micron. Samples of 2 GB HMC (stack of 4 memory dies, each of 4 Gbit) are packed in a 31×31 mm package and have 4 HMC links. Other samples from 2013 have only two HMC links and a smaller package: 16×19.5 mm.

The second version of the HMC specification was published on 18 November 2014 by HMCC. HMC2 offers a variety of SerDes rates ranging from 12.5 Gbit/s to 30 Gbit/s, yielding an aggregate link bandwidth of 480 GB/s (240 GB/s each direction), though promising only a total DRAM bandwidth of 320 GB/sec. A package may have either 2 or 4 links (down from the 4 or 8 in HMC1), and a quarter-width option is added using 4 lanes.

The first processor to use HMCs was the Fujitsu SPARC64 XIfx, which is used in the Fujitsu PRIMEHPC FX100 supercomputer introduced in 2015.

JEDEC's Wide I/O and Wide I/O 2 are seen as the mobile computing counterparts to the desktop/server oriented HMC in that both involve 3D die stacks.
In August 2018, Micron announced a move away from HMC to pursue competing high-performance memory technologies such as GDDR6 and HBM.

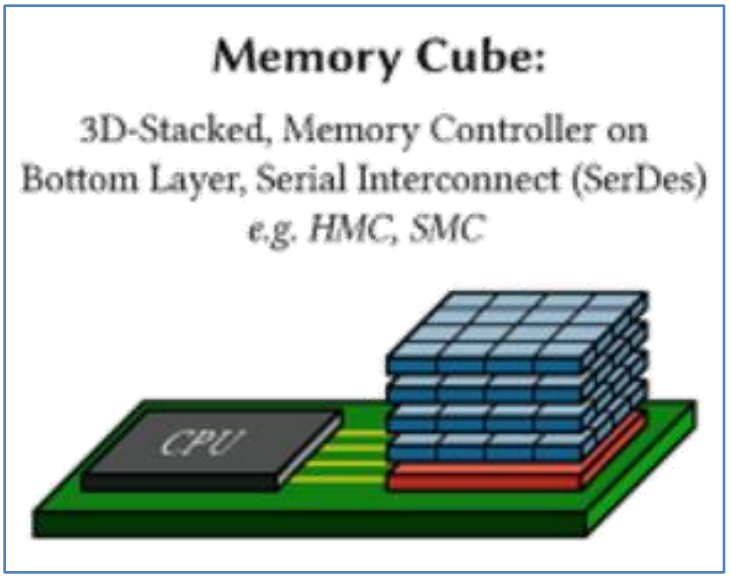


Figure 2. 3: Memory Cube

2.1.2 HMC VS other technologies

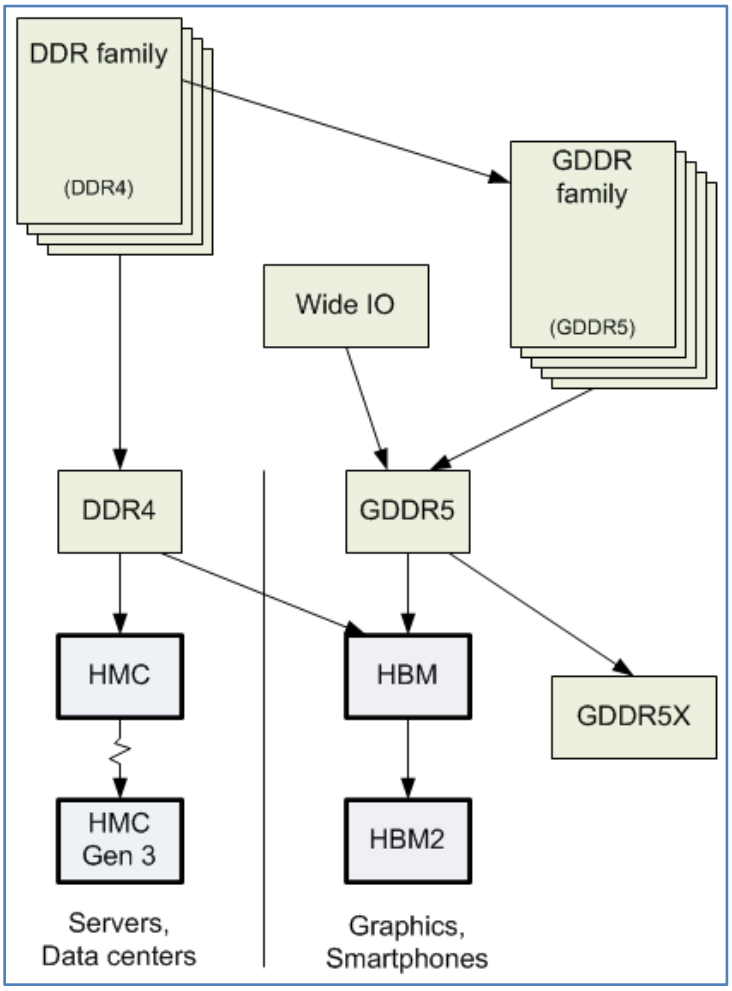


Figure 2. 4: HBM vs. HMC

When compared to other memory technologies, Hybrid Memory Cube (HMC) offers several advantages, but also has some limitations. Here are some comparisons:

Compared to DDR memory: DDR memory is the most widely used type of memory in modern computing systems. DDR memory is known for its low cost and high capacity, but it has limited bandwidth and high-power consumption. In contrast, HMC offers much higher bandwidth and lower power consumption, but at a higher cost. Additionally, HMC has a smaller form factor than DDR memory, which makes it ideal for use in space-constrained applications.

Compared to GDDR memory: GDDR memory is a high-performance memory technology that is commonly used in graphics cards and gaming systems. GDDR memory offers higher bandwidth than DDR memory, but at a higher cost and power consumption. In comparison, HMC offers even higher bandwidth than GDDR memory, with lower power consumption and a smaller form factor.

Compared to High Bandwidth Memory (HBM): HBM is a high-performance memory technology that is similar to HMC in many ways. HBM uses a similar 3D-stacked architecture and TSVs to achieve high bandwidth and low power consumption. However, HBM has a higher memory capacity per chip than HMC, which makes it more suitable for applications that require large amounts of memory.

A big high-level difference: As a graphics-oriented standard, HBM has an architecture that can include a processor (often a GPU) in the same package, providing a tightly coupled high-speed processing unit. HMC devices, on the other hand, are intended to be connected up to a processor (or "host") with the ability to "chain" multiple memories for higher capacity. More on those bits in a moment.

Both have an architectural notion that separates the logic of a controller and the actual bit cells themselves. Putting the logic on a separate chip place it on a standard CMOS process, without all the special memory stuff. But HBM further splits that logic into "device" logic and "host" logic, with the difference largely being the level of abstraction. Host logic interprets commands sent by a processor; device logic largely does the work that DDR logic performs – except that it's no longer on the same die as the memory.

Part of the logic to this is that the memory stack itself will be made by a memory company; the combination with a GPU or CPU might be done by a different company. So, this separates out the logic that is more intimate with the memory from the higher-level logic that drives the memory.

Since the original publication, there's been some confusion that I should clear up, and it depends on your view of what constitutes HBM, mostly, but also with HMC.

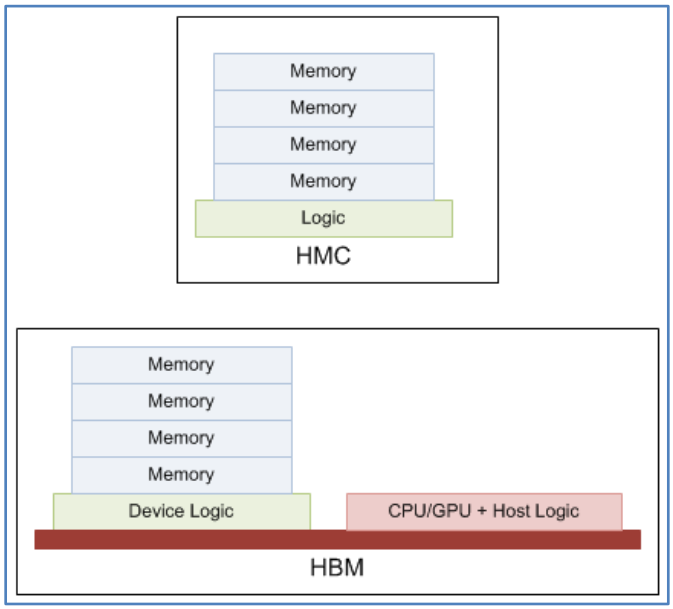


Figure 2. 5: HBM vs. HMC

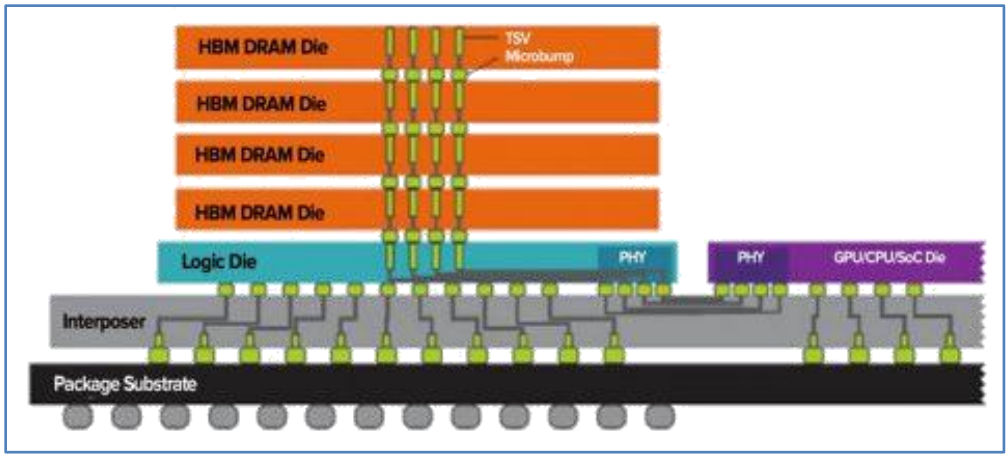


Figure 2. 6: High Bandwidth Memory (HBM)

Overall, HMC offers higher bandwidth, lower power consumption, and a smaller form factor than many other memory technologies. However, it may not be the best choice for applications that require a large amount of memory capacity per chip. Ultimately, the choice of memory technology depends on the specific needs of the application.

2.2 HMC Architecture

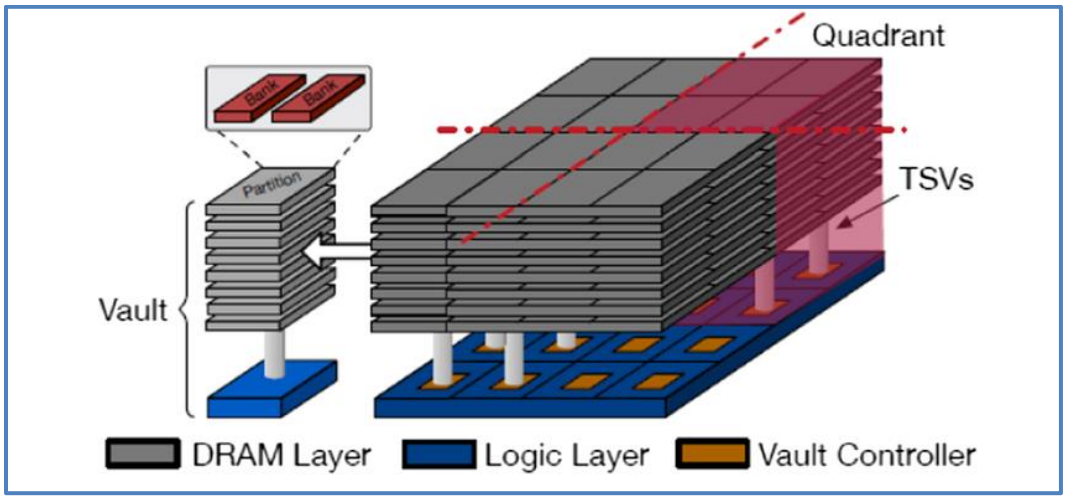


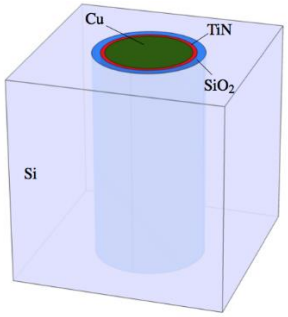
Figure 2. 7: HMC Architecture

A hybrid memory cube (HMC) is a single package containing either four- or eight-DRAM die, and one logic die, all stacked together using through-silicon via (TSV) technology. Within an HMC, memory is organized into vaults. Each vault is functionally and operationally independent.

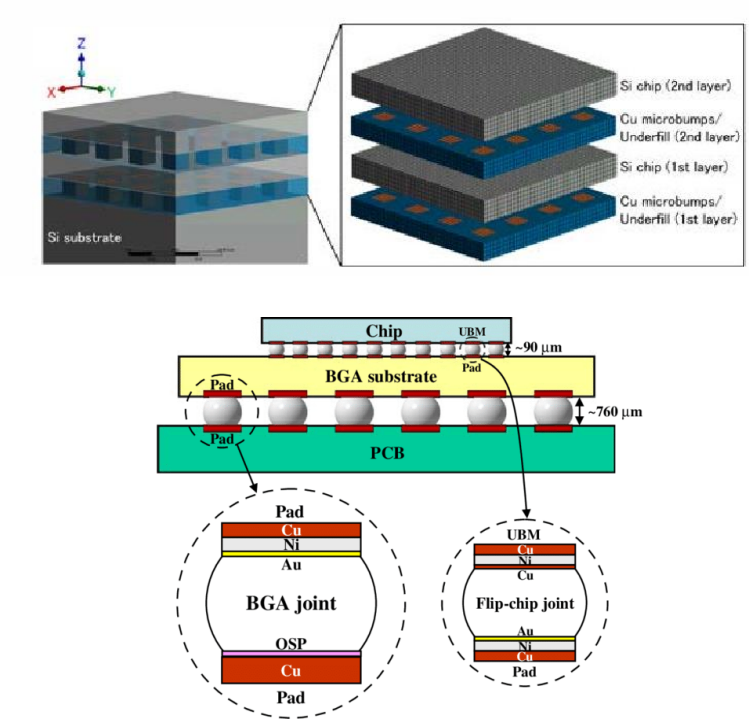
Each vault has a memory controller in the logic base (called a vault controller) that manages all memory reference operations within that vault. Each vault controller determines its own timing requirements. Refresh operations are controlled by the vault controller, eliminating this function from the host memory controller.

Each vault controller may have a queue that is used to buffer references for that vault's memory. The vault controller may execute references within that queue based on need rather than order of arrival. Therefore, responses from vault operations back to the external serial I/O links will be out of order. However, requests from a single external serial link to the same vault/bank address are executed in order. Requests from different external serial links to the same vault/bank address are not guaranteed to be executed in a specific order and must be managed by the host controller. It integrates all DRAM-related management circuits and therefore off-loads the user from DRAM timings.

A single HMC features up to 4 serial links each running with up to 16 lanes and 15 Gb/s per lane. Transactions are packetized instead of using dedicated data and address strobes.



TSV Approach



Micro Bump Approaches

2.3 HMC Configuration

	Configurations	
Number of links in package	4,8	
Memory banks	128	
Number of vaults	16	
Maximum vault data bandwidth	10 GB/s (80 Gb/s)	
	4-link	8-link
Link speed (Gb/s)	10, 12.5, 15	10
Memory density	2,4 GB	4,8 GB
Maximum aggregate link bandwidth[1]	160 GB/s, 200 GB/s, 240 GB/s	320 GB/s

Table 2. 1: HMC Configuration

Note: 1. Link bandwidth includes data as well as packet header and tail. Full width (16 input and 16 output lanes) configuration is assumed.

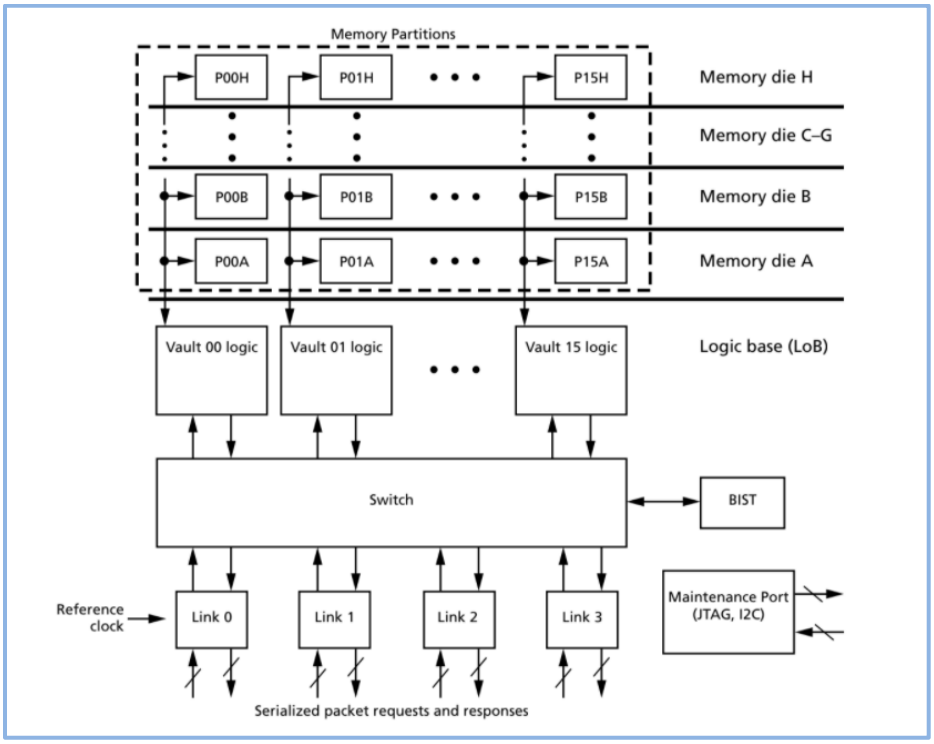


Figure 2. 8: HMC Block Diagram Example Implementation (4-link HMC configuration)

Parameter	Description	Values
FPW	Width of the datapath in FLITs Per Word	2 (256 Bit datapath) 4 (512 Bit datapath) 6 (768 Bit datapath) 8 (1024 Bit datapath)
LOG_FPW	Logarithm of FPW	1 (for FPW=2) 2 (for FPW=4) 3 (for FPW=6/8)
LOG_NUM _LANES	Logarithm of the number of lanes.	3 (half-width Link) 4 (full-width Link)

Figure 2. 9: Datapath and Width Configuration Parameters

2.4 Memory Addressing

A request packet header includes a address field of 34 bits for internal memory addressing within the HMC. This includes vault, bank, and DRAM address bits. The 4-link configurations currently defined will provide a total of up to 4GB of addressable memory locations within one HMC. This requires the lower 32 address bits of the 34-bit field; the upper 2 bits are for future expansion and are ignored by the HMC. The 8-link configurations currently defined will provide a total of up to 8GB of addressable memory locations within one HMC. This requires the lower 33 address bits of the 34-bit field; the upper bit is for future expansion and is ignored by the HMC.

The user can select a specific address mapping scheme so that access of the HMC vaults and banks is optimized for the characteristics of the request address stream. If the request address stream is either random or generally sequential from the low-order address bits, the host can choose to use one of the default address map modes offered in the HMC (See Table, “Default Address Map Mode Table,”). The default address mapping is based on the maximum block size chosen in the address map mode register. It maps the vault address toward the less significant address bits, followed by the bank address just above the vault address. This address mapping algorithm is referred to as “low interleave,” and forces sequential addressing to be spread across different vaults and then across different banks within a vault, thus avoiding bank conflicts. A request stream that has a truly random address pattern is not sensitive to the specific method of address mapping.

2.4.1 Memory Addressing Granularity

The HMC supports READ and WRITE data block accesses with 16-byte granularity from 16B to the value of the maximum block size setting (32B, 64B, or 128B). The maximum block size is set with the Address Configuration register. The maximum block size specified in the address map mode register determines the number of bits within the byte address field, of which the four LSBs are ignored due to the 16-byte granularity (an exception is the BIT WRITE

command). The 4 LSBs of the byte address may be used for future density expansion if 16B block aligned access is preserved. The upper bits of the byte address indicate the starting 16-byte boundary when accessing a portion or all of the block. The vault controller uses these upper bits of the byte address as part of its DRAM column addressing. If the starting 16- byte address in conjunction with the data access length of the request causes the access to go past the end of the maximum block size boundary, the DRAM column addressing will wrap within the maximum block size and continue. For example, if the maximum block size is 64 bytes and a 48-byte READ command is received starting at byte 32, the DRAM access will occur starting at byte 32 up to byte 63, then continue at byte 0 up to byte 15.

2.4.2 Memory Address-to-Link Mapping

Each link is associated with four local vaults that are referred to as a quadrant. Access from a link to a local quadrant may have lower latency than to a link outside the quadrant. Bits within the vault address designate both the specific quadrant and the vaults within that quadrant. See Table for the specific vault addressing of 4-link and 8-link devices.

Address	Description	4-Link Configuration	8-Link Configuration
Byte address	Bytes within the maximum supported block size	The 4 LSBs of the byte address are ignored for READ and WRITE requests (except for BIT WRITE command; “BIT WRITE Command”)	The 4 LSBs of the byte address are ignored for READ and WRITE requests (with the exception of BIT WRITE command; “BIT WRITE Command”)
Vault address	Addresses vaults within the HMC	Lower 2 bits of the vault address specifies 1 of 4 vaults within the logic chip quadrant	Lower 2 bits of the vault address specifies 1 of 4 vaults within the logic chip quadrant
		Upper 2 bits of the vault address specifies 1 of 4 quadrants	Upper 3 bits of the vault address specifies 1 of 8 quadrants
Bank address	Addresses banks within a vault	4GB HMC: addresses 1 of 16 banks in the vault	8GB HMC: addresses 1 of 16 banks in the vault
		2GB HMC: addresses 1 of 8 banks in the vault	4GB HMC: addresses 1 of 8 banks in the vault
DRAM address	Addresses DRAM rows and column within a bank	The vault controller breaks the DRAM address into row and column addresses, addressing 1Mb blocks of 16 bytes each	The vault controller breaks the DRAM address into row and column addresses, addressing 1Mb blocks of 16 bytes each

Table 2. 2: Memory Address-to-Link Mapping

Request Address Bit	2GB – 4 Link Device			4GB – 4 Link Device		
	32-Byte Max Block Size	64-Byte Max Block Size	128-Byte Max Block Size	32-Byte Max Block Size	64-Byte Max Block Size	128-Byte Max Block Size
33	Ignored	Ignored	Ignored	Ignored	Ignored	Ignored
32	Ignored	Ignored	Ignored	Ignored	Ignored	Ignored
31	Ignored	Ignored	Ignored	DRAM[19]	DRAM[19]	DRAM[19]
30	DRAM[19]	DRAM[19]	DRAM[19]	DRAM[18]	DRAM[18]	DRAM[18]
29	DRAM[18]	DRAM[18]	DRAM[18]	DRAM[17]	DRAM[17]	DRAM[17]
28	DRAM[17]	DRAM[17]	DRAM[17]	DRAM[16]	DRAM[16]	DRAM[16]
27	DRAM[16]	DRAM[16]	DRAM[16]	DRAM[15]	DRAM[15]	DRAM[15]
26	DRAM[15]	DRAM[15]	DRAM[15]	DRAM[14]	DRAM[14]	DRAM[14]
25	DRAM[14]	DRAM[14]	DRAM[14]	DRAM[13]	DRAM[13]	DRAM[13]
24	DRAM[13]	DRAM[13]	DRAM[13]	DRAM[12]	DRAM[12]	DRAM[12]
23	DRAM[12]	DRAM[12]	DRAM[12]	DRAM[11]	DRAM[11]	DRAM[11]
22	DRAM[11]	DRAM[11]	DRAM[11]	DRAM[10]	DRAM[10]	DRAM[10]
21	DRAM[10]	DRAM[10]	DRAM[10]	DRAM[9]	DRAM[9]	DRAM[9]
20	DRAM[9]	DRAM[9]	DRAM[9]	DRAM[8]	DRAM[8]	DRAM[8]
19	DRAM[8]	DRAM[8]	DRAM[8]	DRAM[7]	DRAM[7]	DRAM[7]
18	DRAM[7]	DRAM[7]	DRAM[7]	DRAM[6]	DRAM[6]	DRAM[6]
17	DRAM[6]	DRAM[6]	DRAM[6]	DRAM[5]	DRAM[5]	DRAM[5]
16	DRAM[5]	DRAM[5]	DRAM[5]	DRAM[4]	DRAM[4]	DRAM[4]
15	DRAM[4]	DRAM[4]	DRAM[4]	DRAM[3]	DRAM[3]	DRAM[3]
14	DRAM[3]	DRAM[3]	DRAM[3]	DRAM[2]	DRAM[2]	Bank[3]
13	DRAM[2]	DRAM[2]	Bank[2]	DRAM[1]	Bank[3]	Bank[2]
12	DRAM[1]	Bank[2]	Bank[1]	Bank[3]	Bank[2]	Bank[1]
11	Bank[2]	Bank[1]	Bank[0]	Bank[2]	Bank[1]	Bank[0]
10	Bank[1]	Bank[0]	Vault[3]	Bank[1]	Bank[0]	Vault[3]
9	Bank[0]	Vault[3]	Vault[2]	Bank[0]	Vault[3]	Vault[2]
8	Vault[3]	Vault[2]	Vault[1]	Vault[3]	Vault[2]	Vault[1]
7	Vault[2]	Vault[1]	Vault[0]	Vault[2]	Vault[1]	Vault[0]
6	Vault[1]	Vault[0]	Byte[6], DRAM[2]	Vault[1]	Vault[0]	Byte[6], DRAM[2]
5	Vault[0]	Byte[5], DRAM[1]	Byte[5], DRAM[1]	Vault[0]	Byte[5], DRAM[1]	Byte[5], DRAM[1]
4	Byte[4], DRAM[0]	Byte[4], DRAM[0]	Byte[4], DRAM[0]	Byte[4], DRAM[0]	Byte[4], DRAM[0]	Byte[4], DRAM[0]
3	Byte[3] = Ignored	Byte[3] = Ignored	Byte[3] = Ignored	Byte[3] = Ignored	Byte[3] = Ignored	Byte[3] = Ignored
2	Byte[2] = Ignored	Byte[2] = Ignored	Byte[2] = Ignored	Byte[2] = Ignored	Byte[2] = Ignored	Byte[2] = Ignored
1	Byte[1] = Ignored	Byte[1] = Ignored	Byte[1] = Ignored	Byte[1] = Ignored	Byte[1] = Ignored	Byte[1] = Ignored
0	Byte[0] = Ignored	Byte[0] = Ignored	Byte[0] = Ignored	Byte[0] = Ignored	Byte[0] = Ignored	Byte[0] = Ignored

Figure 2. 10: Default Address-Map Mode Table – 4-link Devices

Request Address Bit	4GB – 8 Link Device			8GB – 8 Link Device		
	32-Byte Max Block Size	64-Byte Max Block Size	128-Byte Max Block Size	32-Byte Max Block Size	64-Byte Max Block Size	128-Byte Max Block Size
33	Ignored	Ignored	Ignored	Ignored	Ignored	Ignored
32	Ignored	Ignored	Ignored	DRAM[19]	DRAM[19]	DRAM[19]
31	DRAM[19]	DRAM[19]	DRAM[19]	DRAM[18]	DRAM[18]	DRAM[18]
30	DRAM[18]	DRAM[18]	DRAM[18]	DRAM[17]	DRAM[17]	DRAM[17]
29	DRAM[17]	DRAM[17]	DRAM[17]	DRAM[16]	DRAM[16]	DRAM[16]
28	DRAM[16]	DRAM[16]	DRAM[16]	DRAM[15]	DRAM[15]	DRAM[15]
27	DRAM[15]	DRAM[15]	DRAM[15]	DRAM[14]	DRAM[14]	DRAM[14]
26	DRAM[14]	DRAM[14]	DRAM[14]	DRAM[13]	DRAM[13]	DRAM[13]
25	DRAM[13]	DRAM[13]	DRAM[13]	DRAM[12]	DRAM[12]	DRAM[12]
24	DRAM[12]	DRAM[12]	DRAM[12]	DRAM[11]	DRAM[11]	DRAM[11]
23	DRAM[11]	DRAM[11]	DRAM[11]	DRAM[10]	DRAM[10]	DRAM[10]
22	DRAM[10]	DRAM[10]	DRAM[10]	DRAM[9]	DRAM[9]	DRAM[9]
21	DRAM[9]	DRAM[9]	DRAM[9]	DRAM[8]	DRAM[8]	DRAM[8]
20	DRAM[8]	DRAM[8]	DRAM[8]	DRAM[7]	DRAM[7]	DRAM[7]
19	DRAM[7]	DRAM[7]	DRAM[7]	DRAM[6]	DRAM[6]	DRAM[6]
18	DRAM[6]	DRAM[6]	DRAM[6]	DRAM[5]	DRAM[5]	DRAM[5]
17	DRAM[5]	DRAM[5]	DRAM[5]	DRAM[4]	DRAM[4]	DRAM[4]
16	DRAM[4]	DRAM[4]	DRAM[4]	DRAM[3]	DRAM[3]	DRAM[3]
15	DRAM[3]	DRAM[3]	DRAM[3]	DRAM[2]	DRAM[2]	Bank[3]
14	DRAM[2]	DRAM[2]	Bank[2]	DRAM[1]	Bank[3]	Bank[2]
13	DRAM[1]	Bank[2]	Bank[1]	Bank[3]	Bank[2]	Bank[1]
12	Bank[2]	Bank[1]	Bank[0]	Bank[2]	Bank[1]	Bank[0]
11	Bank[1]	Bank[0]	Vault[4]	Bank[1]	Bank[0]	Vault[4]
10	Bank[0]	Vault[4]	Vault[3]	Bank[0]	Vault[4]	Vault[3]
9	Vault[4]	Vault[3]	Vault[2]	Vault[4]	Vault[3]	Vault[2]
8	Vault[3]	Vault[2]	Vault[1]	Vault[3]	Vault[2]	Vault[1]
7	Vault[2]	Vault[1]	Vault[0]	Vault[2]	Vault[1]	Vault[0]
6	Vault[1]	Vault[0]	Byte[6], DRAM[2]	Vault[1]	Vault[0]	Byte[6], DRAM[2]
5	Vault[0]	Byte[5], DRAM[1]	Byte[5], DRAM[1]	Vault[0]	Byte[5], DRAM[1]	Byte[5], DRAM[1]
4	Byte[4], DRAM[0]	Byte[4], DRAM[0]	Byte[4], DRAM[0]	Byte[4], DRAM[0]	Byte[4], DRAM[0]	Byte[4], DRAM[0]
3	Byte[3] = Ignored	Byte[3] = Ignored	Byte[3] = Ignored	Byte[3] = Ignored	Byte[3] = Ignored	Byte[3] = Ignored
2	Byte[2] = Ignored	Byte[2] = Ignored	Byte[2] = Ignored	Byte[2] = Ignored	Byte[2] = Ignored	Byte[2] = Ignored
1	Byte[1] = Ignored	Byte[1] = Ignored	Byte[1] = Ignored	Byte[1] = Ignored	Byte[1] = Ignored	Byte[1] = Ignored
0	Byte[0] = Ignored	Byte[0] = Ignored	Byte[0] = Ignored	Byte[0] = Ignored	Byte[0] = Ignored	Byte[0] = Ignored

Figure 2. 11: Default Address-Map Mode Table – 8-link Devices

2.4.3 Address Mapping Mode Register

The address mapping mode register provides user control of mapping portions of the ADRS field within the header of request packets to the vault and bank addresses. When using default address-mapping, the address mapping mode field should be set based on specific system requirements. In addition to the default address mapping algorithms, user-defined fields are provided for the user to specify a unique mapping algorithm. There is a user-defined field for the vault address and the bank address. Each field holds a 5-bit encoded value pointing to the corresponding bit position in the ADRS field of the request packet header. This represents the LSB that will be used for the vault or bank address, with the more significant bits of the sub address coming from the bit positions immediately above it in the request header ADRS field. When operating with the user-defined modes, it is the user's responsibility to set the user-defined encoded fields so that the vault and bank addresses do not overlap each other or overlap the byte address within the request header ADRS field. Note that the byte address is always fixed at the LSBs of the request header ADRS field. The user must also be aware that the number of bits in the bank address is dependent upon the size of the HMC. The remaining bits of the request ADRS field not specified as the vault and bank addresses become the DRAM address. The address map mode register is writable via the MODE WRITE command or via the maintenance interface (I2C or JTAG bus). Note that because the MAX block size is defaulted to 64 bytes, the host boot block code must either be compatible (issue requests of 64 bytes or smaller) or issue a MODE WRITE command to change the MAX block size.

2.4.4 DRAM Addressing

Each bank in the HMC contains 16,777,216 memory bytes (16 MB). A WRITE or READ command to a bank accesses 32 bytes of data for each column fetch. The bank configuration is identical in all specified HMC configurations.

Chapter 3 : Hybrid Memory Cube Controller

3.1 What is Memory Controller

Before diving deeply into Hybrid Memory Cube Controller , Memory controller generally should be defined. A memory controller is a critical component in computer systems that manages the flow of data between the central processing unit (CPU) and the memory subsystem. It plays a crucial role in ensuring efficient data transfer, data integrity , and overall system performance.

3.2 what is Hybrid Memory Cube Controller

Hybrid Memory Cube (HMC) is a type of high-performance memory technology developed by the Hybrid Memory Cube Consortium. It offers significant advantages over traditional memory architectures such as DDR (Double Data Rate) SDRAM (Synchronous Dynamic Random-Access Memory) in terms of bandwidth, power efficiency, and physical footprint . The HMC controller is responsible for controlling and managing the communication between the CPU and the Hybrid Memory Cube. It handles tasks such as data transfers, addressing, error correction, and power management. The controller acts as an interface between the CPU and the HMC, providing a bridge for data exchange.

It provides some additional key features than traditional memory controller:

1. *Increased Bandwidth:* HMC provides much higher bandwidth compared to traditional memory technologies. The controller maximizes the utilization of available bandwidth by efficiently scheduling data transfers and optimizing memory access patterns.
2. *Reduced Power Consumption:* HMC controllers employ advanced power management techniques to minimize power consumption. This includes features such as link power management, voltage scaling, and power gating.
3. *Enhanced Scalability:* HMC supports high levels of scalability, allowing for multiple memory cubes to be stacked together. The controller handles the management of multiple cubes and enables efficient data transfers across the stacked memory devices.
4. *Improved Reliability:* HMC controllers incorporate advanced error correction mechanisms to ensure data integrity and reliability. They detect and correct errors that may occur during data transmission, thereby improving system stability.

5. *Advanced Features:* HMC controllers may include additional features such as built-in self-test (BIST) capabilities, diagnostic tools , and performance monitoring . These features aid in system debugging, testing and optimization.

3.3 HMC Controller Architecture

3.3.1 Logical view of HMC Controller

The host controller '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 '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.

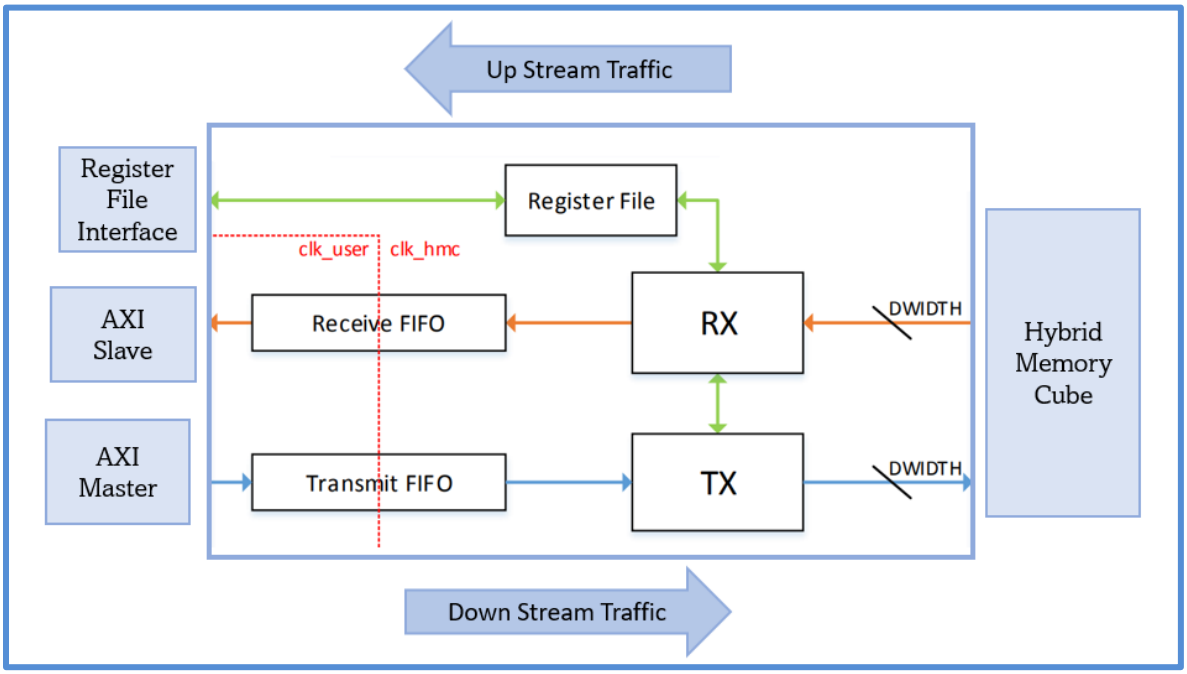


Figure 3. 1: openHMC Controller Top Module

3.3.2 Detailed view of HMC Controller Architecture

3.3.2.1 Asynchronous TX/RX FIFOS

The asynchronous FIFOs connect the user logic in the clock user clock domain to the HMC controller in the clock HMC clock domain. In other words, these FIFOs is to match two clocks.

3.3.2.1 Transmitter Link

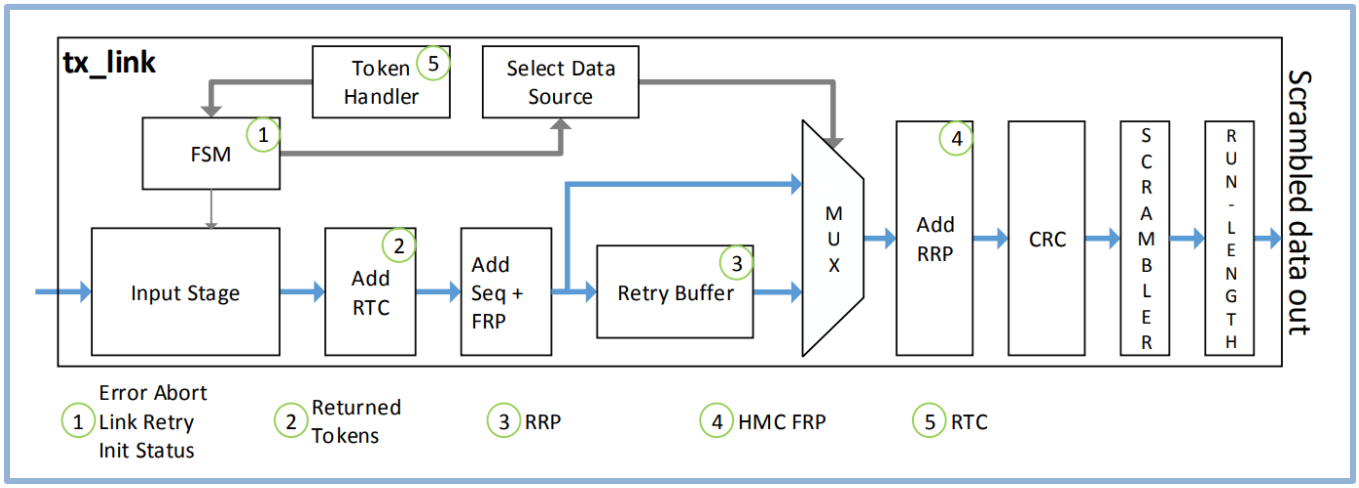


Figure 3. 2: TX Link Diagram

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 data is retransmitted out of the retry buffer instead of the regular data path. Eventually the Return Retry Pointer (RRP) which is the last received HMC FRP is added, the CRC generated, and data is scrambled and reordered on a lane-by-lane basis.

- **Retry Buffer**

  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 therefore not saved in the retry buffer. The retry buffer consists of FPW. times 128-bit RAMs so that each FLIT can be addressed independently. One address, which is also 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).

- **Scrambler**

  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/10.Scramblers aid in clock recovery at the receiver end. By introducing signal transitions and randomization, the receiver's clock recovery circuitry can extract the clock signal from the received data more easily . Clock recovery is crucial for proper data synchronization and accurate retrieval of transmitted information . Scramblers can help improve the system's resistance to certain types of noise . By spreading the energy of the transmitted signal across a wider frequency spectrum, scramblers can make the signal less susceptible to interference caused by specific frequency components or narrowband noise sources.

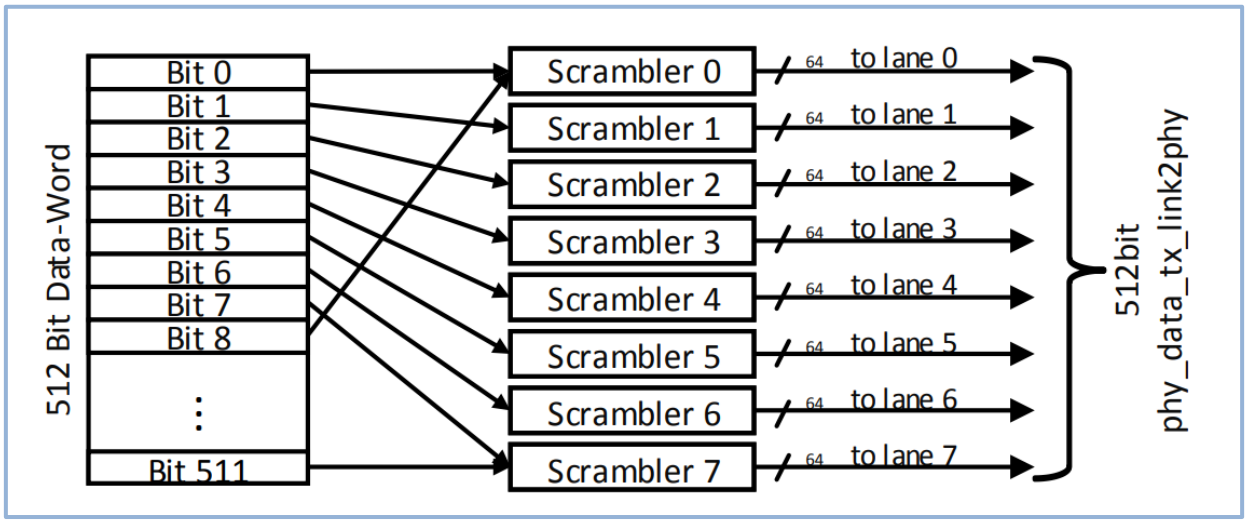


Figure 3. 3: Data-Reordering: 4FLIT/512bit example

- **Lane Run Length Limiter**

  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.

3.3.2.2 Receiver Link

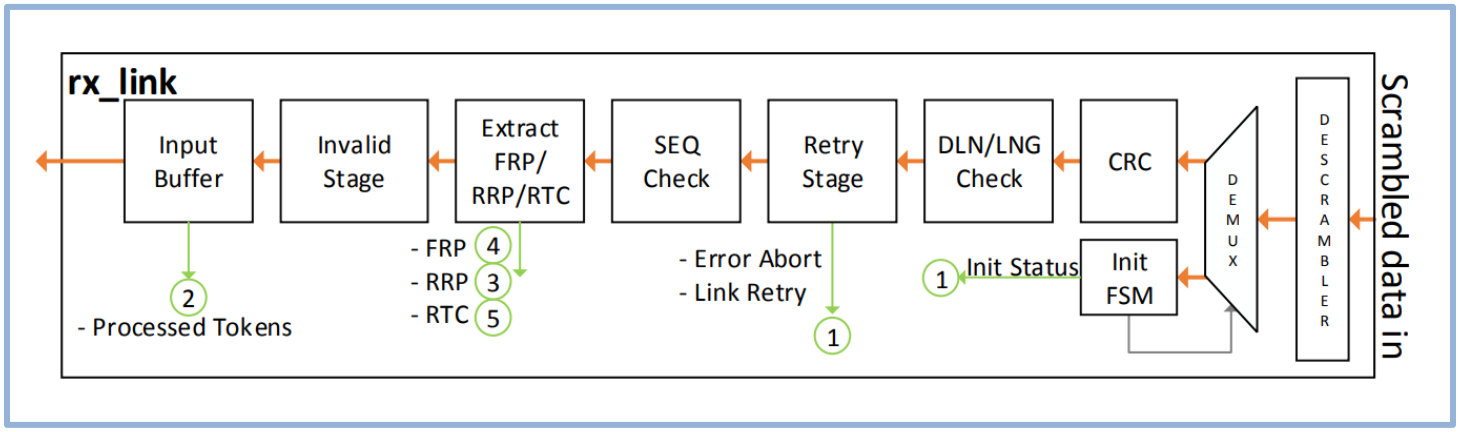


Figure 3. 4: RX Link Diagram

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 passed all checks will enter the input buffer and can be collected at the AXI-4 slave interface. Figure 3.4 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 data path is only selected after link initialization is done. For this purpose, the initialization FSM controls a Multiplexer (MUX) to distribute input data.

- **Receiver descrambler**

  The 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 locked. Additionally, each descrambler expects a dedicated, so called “bit_slip” single 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.

3.4 Some main topics used in the Design

3.4.1 Advanced eXtensible Interface "AXI"

The Advanced eXtensible Interface (AXI4) protocol was developed by ARM as part of the AMBA (Advanced Microcontroller Bus Architecture) specification. The AXI4 protocol is a widely used communication protocol in the field of digital design, especially for connecting components within a System-on-Chip (SoC) or an FPGA design.

The AXI4 protocol features are:

- Suitable for high-bandwidth and low-latency designs.
- High-frequency operation is provided without using complex bridges.
- protocol meets the interface requirements of a wide range of components.
- Suitable for memory controllers with high initial access latency.
- Flexibility in the implementation of interconnect architectures is provided.

3.4.1.1 Types of AXI4 Interfaces

There are Three Types of AXI4 Interfaces:

- Full AXI4 - High-performance communication, using memory-mapped addresses.
- AXI-Lite - Lightweight and simple memory-mapped interface, used for single transaction communication.
- AXI4-Stream - 'Direct' device communication, removing the need for addresses and allowing for maximum data transfer.

The AXI4-Stream protocol is used as a common interface to connect components that want to exchange data. The interface can be used to connect a single master, which generates data, to a single slave, which receives data. The protocol can also be used to connect larger numbers of master and slave components.

3.4.1.2 The AXI4-Stream protocol features

- It is simpler and more lightweight compared to other types of AXI interfaces.
- It does not support burst or interleaved transactions. It focuses on the transfer of streaming data one element at a time.
- The Data signal width is fixed and defined by the Data signal width. Every data element is sent in a single cycle.
- It uses the valid and ready signals for flow control between the master and slave.

3.4.1.3 The AXI4 Stream Applications:

- It is used in FPGAs to interface with custom IP cores and accelerator blocks.
- It is used in digital signal processing (DSP) applications, such as audio processing and signal analysis.
- It is used in high-performance computing (HPC) applications.
- It is used in real-time control systems like robotics and automotive applications.

3.4.1.4 The AXI4-Stream Signals:

The signals used in the AXI4 Stream interface are :

- TVALID : this signal indicates the validity of the data. When this signal is asserted, it indicates that the data on the interface is valid .
- TREADY: this signal indicates the capability of the slave to receive data . When this signal is asserted, it indicates that the slave is ready to receive data from the master.
- TDATA: this signal contains the streaming data that is passing across the interface.
- TUSER : this signal is user-defined sideband information that can be transmitted along the data stream.

3.4.1.5 Handshake process

The TVALID and TREADY handshake determines when information is passed across the interface. A two-way flow control mechanism enables both the master and slave to control the rate at which the data and control information is transmitted across the interface. For a transfer to occur both the TVALID and TREADY signals must be asserted.

A master is not permitted to wait until TREADY is asserted before asserting TVALID. Once TVALID is asserted it must remain asserted until the handshake occurs.

A slave is permitted to wait for TVALID to be asserted before asserting the corresponding TREADY.

- **Handshake with TVALID and TREADY asserted simultaneously**

In the same cycle of clk_user, the master asserts TVALID high and the slave asserts TREADY high. In this case, Transfer occurs in the same cycle of clk_user, as shown in Figure 3.5.

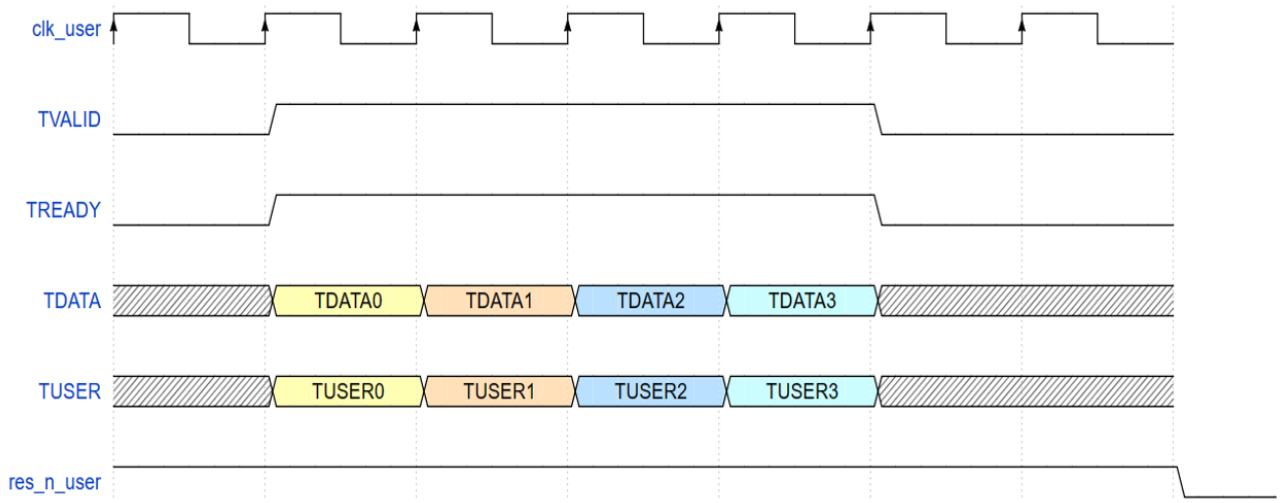


Figure 3. 5: Handshake with TVALID and TREADY asserted simultaneously

- **Handshake with TREADY asserted before TVALID**

The slave asserts TREADY high before TDATA and TUSER are valid . In this case, the transfer occurs once the master asserts TVALID high, as shown in Figure 3.6.

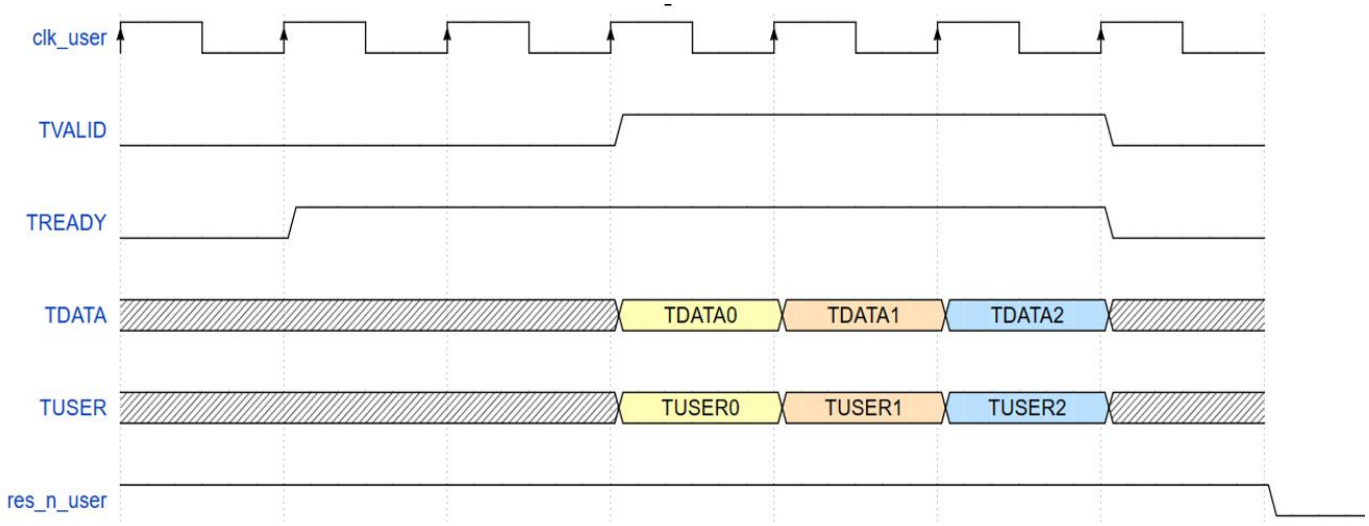


Figure 3. 6: Handshake with TREADY asserted before TVALID

- **Handshake with TVALID asserted before TREADY**

The master asserts TVALID high before the slave asserts TREADY high. TDATA and TUSER must remain unchanged until TREADY is asserted high by the slave. In this case, transfer occurs once the slave asserts TREADY HIGH, as shown in Figure 3.7.

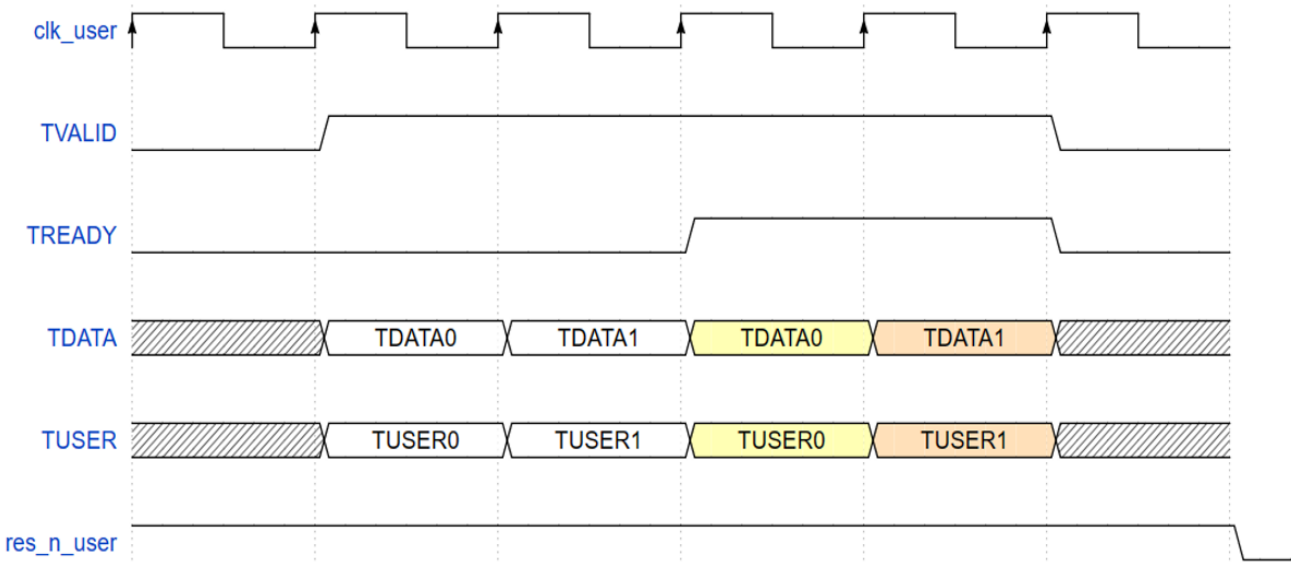


Figure 3. 7: Handshake with TVALID asserted before TREADY

There is no data transfer TREADY and TVALID ‘1’ values never overlap, as shown in Figure 3.8.

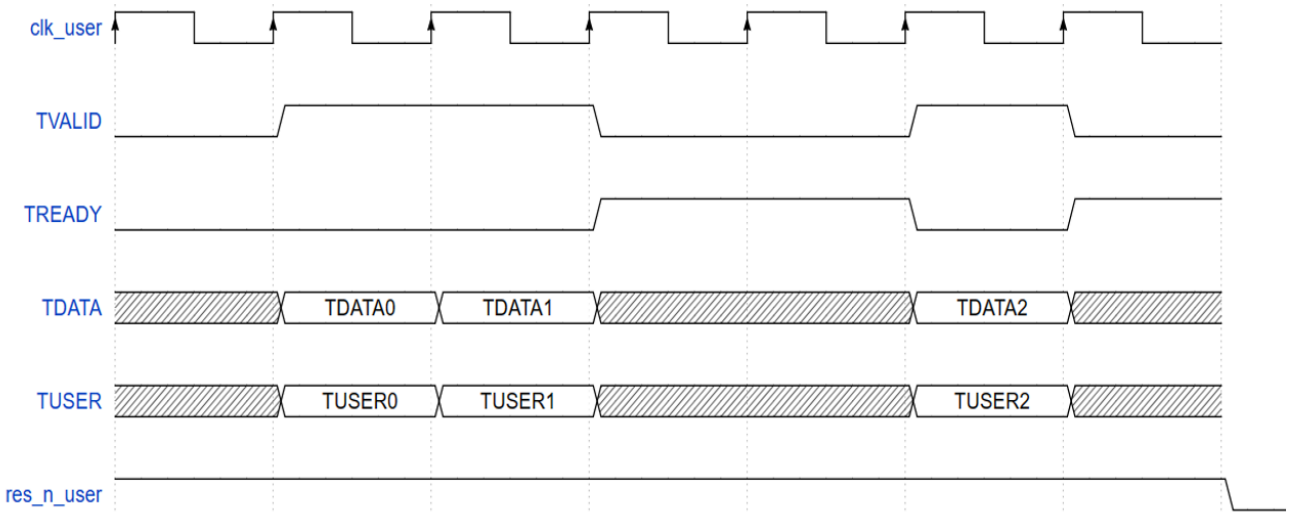


Figure 3. 8: No Data Transfer

3.4.2 CRC

3.4.2.1 CRC Technique

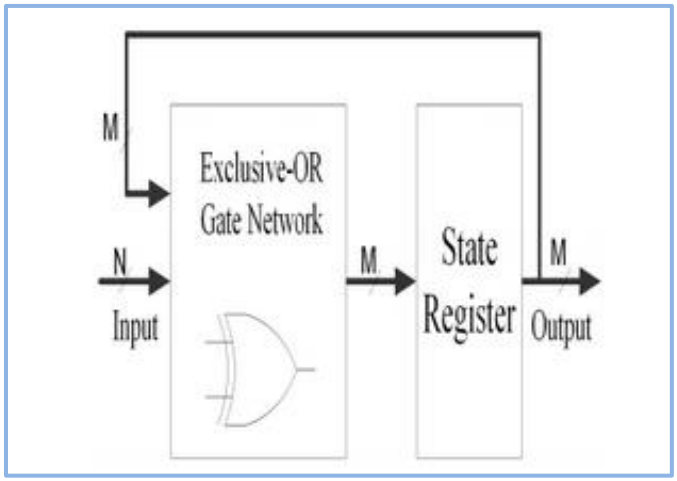


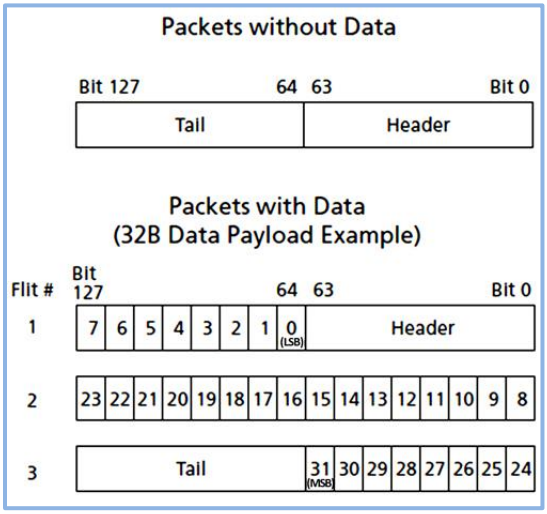
Figure 3. 9: General Architecture of a CRC Computation Circuit

- A channel coding technology that generates short, fixed digit check code according to network data packets or computer files. It is mainly used to detect or check the possible errors after data transmission or storage. It may use the principle of:

1. division (use remainder/ignore quotient) over (Galois Field) GF (2). If the elements of GF(2) are seen as Boolean values, then the addition is the same as that of the logical XOR operation. Since each element equals its opposite, subtraction is thus the same operation as addition.
2. or shift register with polynomial XOR taps to detect errors.

- 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.
- Polynomial formula: The abbreviation of the generated formula, expressed in hexadecimal. Ignore the highest "1"= 32'h741B8CD7
- CRC = (Data Word) XOR (Polynomial formula)

3.4.2.2 CRC in HMC



A CRC field is included in the tail of every packet that covers the entire packet, including the header, all data, and the non-CRC 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. ECC provides error coverage of the data from the vault controller to and from the DRAM arrays. Thus, data is protected along the entire path to and from the memory device. 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:

$$p(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x + 1$$

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.

- **Poisoned Packets**

A poisoned packet is uniquely identified by the fact that its CRC is inverted from the correct CRC value. All packet CRC verification logic should include checking for an inverted value of good CRC. A poisoned packet starts out as a good packet (as identified by its good CRC) but for various reasons can become poisoned. The most common cause of a poisoned packet is a link error. There

are both valid and invalid fields within the header and tail of poisoned packets. Note that receiving a poisoned packet does not trigger a link retry.

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, in this case a vault controller, 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. Another example of a forwarded poisoned packet is the case in which DRAM errors occur and are internally retried. If a parity error occurs on the command or address from the vault controller to the DRAM, a response packet may be generated and transmission back to the requester could be started before the parity error is detected. In this case, the CRC in the tail of the response packet will be poisoned, meaning the vault controller will invert the CRC and insert it into the tail of the packet. Consequently, the requester may receive the poisoned packet and must drop it. The vault controller will retry the DRAM request, and upon successful DRAM access, another response packet will be generated to replace the poisoned one.

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.

3.5 HMC Controller Interfaces

3.5.1 System Interface

The controller top module expects a clock and a reset per clock domain. 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 3.10 shows the system interface. Note that both resets are active low.

Signal	Description
clk_hmc	If SYNC_AXI4_IF is set to 0, source both clocks from the same source
clk_user	Clock of the user. Connected if SYNC_AXI4_IF=0
res_n_hmc	Active low synchronous hmc reset.
res_n_user	Active low synchronous user reset. Connected if SYNC_AXI4_IF=0

Table 3. 1: System Interface Signals

Figure 3. 10: System Interface

3.5.2 Register File Interface

The Register File features three main types of registers: Control, Status, and Counter. Control registers directly affect openHMCs or HMC operation. Status registers can be used to monitor the openHMC status, especially during initialization. Counters allow performance measurement.

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. 2: Register File Address Map

Field Name	Access

link_up	Ro
link_training	Ro
sleep_mode	Ro
FERR_N	Ro
lanes_reversed	Ro
phy_tx_ready	Ro
phy_rx_ready	Ro
hmc_tokens_remaining	Ro
rx_tokens_remaining	Ro
lane _polarity _reversed	Ro

Table 3. 4: Status General

Field Name	Access
lane_descramblers_locked	Ro
descrambler_part_aligned	Ro
descrambler_aligned	Ro
all_descramblers_aligned	Ro
status_init_rx_init_state	Ro
status_init_tx_init_state	Ro

Table 3. 3: Status Init

Field Name	Access
p_rst_n	RW
hmc_init_cont_set	RW
set_hmc_sleep	RW
warm_reset	RW
scrambler_disable	RW
run_length_enable	RW
rx_token_count	RW
irtry_received_threshold	RW
irtry_to_send	RW

Table 3. 5: Control

Field Name	Access
sent_p	RW
sent_np	RW
sent_r	RW
poisoned_packets	RW
rcvd_rsp	RW
counter_reset	RW
tx_link_retries	RW
run_length_bit_flip	RW
error_abort_not_cleared	RW

Table 3. 6: Counter

A Register File module allows to control and monitor the openHMC operation. The interface signals are shown in Figure 3.11 and described in Table 3.6. 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.

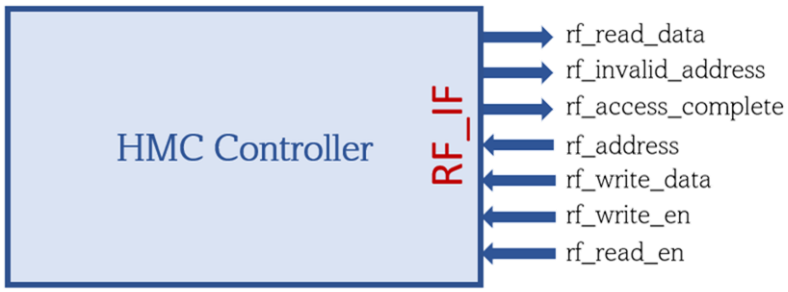


Figure 3. 11: Register File Interface

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

Table 3. 7: Register File Interface Signals

3.5.3 AXI Req Interface

It is used to connect the AXI Req Agent with the HMC Controller.

Figure 3. 12: AXI Request Interface

Signal	Width	Description
TVALID	1 bit	-TVALID indicates that the AXI Request agent is driving a valid data. -TDATA and TUSER are sampled on TX when TVALID=1 and TREADY=1.
TREADY	1 bit	-TREADY indicates that the HMC Controller is ready to receive data (TDATA and TUSER) from the AXI Request agent.
TDATA	FPW*128	TDATA contains the data that is passing across the interface.
TUSER	FPW*16	TUSER is user-defined sideband information that can be transmitted along the data stream. -contains{tail,hdr,valid} - 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

Table 3. 8: AXI Request Interface Signals

3.5.4 HMC Memory Interface (Reactive agent interface)

It is used to connect the controller with the memory cube, or a set of memory cubes. In our case we made a model for the cube as a reactive slave agent to interact with the controller.

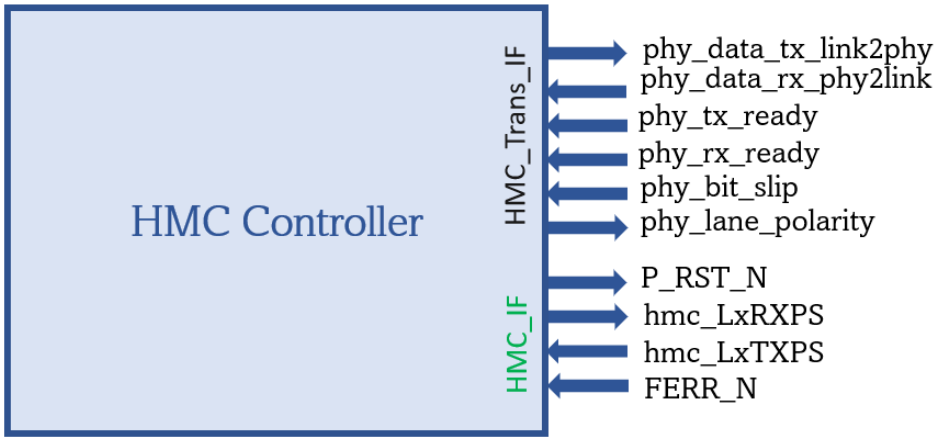


Figure 3. 13: HMC Memory Interface

Signal	Width	Description
phy_data_tx_link2phy	DWIDTH	Lane by lane ordered output
phy_data_rx_phy2link	DWIDTH	Lane by lane ordered input
phy_tx_ready	1	Signalize that the transceivers are ready
phy_rx_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	NUM_LANES	Connect transceiver polarity inputs if polarity is controlled within the transceivers and 'CTRL_LANE_POLARITY' is set to 1.
P_RST_N	1	reset pin for the memory cube.
LxRXPS	1	Indicates that the H.M-Cube went into sleep mode. (Active low)
LxTXPS	1	Indicates that the H.M.C-Controller went into sleep mode and requests the H.M-Cube to change its mode into sleep mode or power reduction mode. (Active low)
FERR_N	1	Active low fatal error indicator.

Table 3. 9: HMC Memory Interface Signals

3.5.5 AXI Rsp Interface

It is used to connect the HMC Controller with AXI Rsp Agent.

Figure 3. 14: AXI Response Interface

Signal	Width	Description
TVALID	1 bit	-TVALID indicates that the HMC Controller is driving a valid data. -TDATA and TUSER are valid when TVALID=1.
TREADY	1 bit	-TREADY indicates that the AXI Rsp Agent is ready to receive data (TDATA and TUSER) from the HMC Controller.
TDATA	FPW*128	TDATA contains the data that is passing across the interface.
TUSER	FPW*16	-TUSER is user-defined sideband information that can be transmitted along the data stream. -contains{tail,hdr,valid} - 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

Table 3. 10: AXI Response Interface Signals

3.6 Main packet types

3.6.1 Flow packet

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 decrease the token count when it transmits flow packets . Flow packets are not considered a request, they do not have a valid tag, and they do not have a corresponding response packet. In General flow packets are used to transfer data between the host device and the HMC. Flow packets use the request packet header and tail layouts described in Figure 3.15 and Figure 3.16.

3.6.2 Transaction packets

There are two types of transaction packets used in HMC Controllers, Request packets: these packets are sent by the host processor to request data from the HMC, and Response packets: these packets are sent by the HMC to return data requested by the host processor.

3.6.2.1 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. Request packet headers for request commands and flow commands contain the fields shown in Table 3.11 and also the tail fields shown in Table 3.12. Specific commands may require some of the fields to be 0.

63 61	60 58	57 24	23 15	14 11	10 7	6	5 0
CUB[2:0]	RES[2:0]	ADRS[33:0]	Tag [10:0]	DLN[3:0]	LNG[3:0]	RES	CMD[5:0]

Figure 3. 15: Request Packet Header Layout

Name	Field Label	Bit Count	Bit Range	Function
Cube ID	CUB	3	[63:61]	CUB field used to match request with target cube. The internal Cube ID Register defaults to the value read on external CUB pins of each HMC device
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 ran
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.

Table 3. 11: Request Packet Header Fields

63 32	31 27	26 24	23 19	18 16	15 8	7 0
CRC[31:0]	RTC[4:0]	SLID[2:0]	RES[4:0]	SEQ [2:0]	FRP[7:0]	RRP[7:0]

Figure 3. 16: Request Packet Tail Layout

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	9	[15:8]	Retry pointer representing this packet's position in the retry buffer
Return retry pointer	RRP	9	[7:0]	Retry pointer being returned for other side of link

Table 3. 12: Request Packet Tail Field

3.6.2.2 Response packet

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. Response packet headers shown in Table 3.13 and also the tail fields shown in Table 3.14.

63 42	41 39	38 33	32 24	23 15	14 11	10 7	6	5 0
RES[21:0]	SLID[2:0]	RES[5:0]	TGA [8:0]	TAG[8:0]	DLN[3:0]	LNG[3:0]	RES	CMD[5:0]

Figure 3. 17: Response Packet Header Layout

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
Source Link ID	SLID	3	[41:39]	Used to identify the source link for response routing. This value is copied from the corresponding Request header and used for response routing purposes. The host can ignore these bits (except for including them in the CRC calculation)
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	TGA	9	[32:15]	Field that can contain a write response tag. Bits are reserved if the function is not supported. (optional)
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 128-bit FLITs. 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

Table 3. 13: Response Packet Tail Layout

63 32	31 27	26 20	19	18 16	15 8	7 0
CRC[31:0]	RTC[4:0]	ERRSTAT[6:0]	DINV	SEQ[2:0]	FRP[7:0]	RRP[7:0]

Figure 3. 18: Response Packet Tail Layout

Name	field label	bit count	bit range	Function
Cyclic redundancy check.	CRC	32	[63:32]	Error-detecting code field that covers the entire packet
Return token counts	RTC	5	[31:27]	Return token count for transaction-layer flow control. In the response packet tail, the RTC contains an encoded value equaling the returned tokens. The tokens represent incremental available space in the HMC input buffer. See the table below for the RTC encoding key.
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 the other side of link.

Table 3. 14: Response Packet Tail Fields

3.6.3 The command Field

We didn't use all commands of the specs. For example, the MODE READ and WRITE (MD_WR) request commands access the internal hardware mode and status registers within the logic layer of the HMC, and here there isn't HMC memory or its register file. This field is responsible for specifying the packet type.

3.6.3.1 Request Commands

All request commands are issued by the requester to the responder.

Symbol	Description	CMD Bit	LNG	Response
WRITE requests				
WR16	16-byte WRITE request	6'b001000	2	WR_RS
WR32	32-byte WRITE request	6'b001001	3	
WR48	48-byte WRITE request	6'b001010	4	
WR64	64-byte WRITE request	6'b001011	5	
WR80	80-byte WRITE request	6'b001100	6	
WR96	96-byte WRITE request	6'b001101	7	
WR112	112-byte WRITE request	6'b001110	8	
WR128	128-byte WRITE request	6'b001111	9	
ATOMIC Requests				
TWO_ADD8	Dual 8-byte add immediate	6'b010010	2	WR_RS
ADD16	Single 16-byte add immediate	6'b010011		
Posted Write Requests				
P_WR16	16-byte POSTED WRITE request	6'b011000	2	None
P_WR32	32-byte POSTED WRITE request	6'b011001	3	
P_WR48	48-byte POSTED WRITE request	6'b011010	4	
P_WR64	64-byte POSTED WRITE request	6'b011011	5	
P_WR80	80-byte POSTED WRITE request	6'b011100	6	
P_WR96	96-byte POSTED WRITE request	6'b011101	7	
P_WR112	112-byte POSTED WRITE request	6'b011110	8	
P_WR128	128-byte POSTED WRITE request	6'b011111	9	
POSTED ATOMIC Requests				
P_TWO_ADD8	Posted dual 8-byte add immediate	6'b100010	2	None
P_ADD16	Posted single 16-byte add immediate	6'b100011		
READ Requests				
RD16	16-byte READ request	6'b110000	1	RD_RS
RD32	32-byte READ request	6'b110001		
RD48	48-byte READ request	6'b110010		
RD64	64-byte READ request	6'b110011		
RD80	80-byte READ request	6'b110100		
RD96	96-byte READ request	6'b110101		
RD112	112-byte READ request	6'b110110		
RD128	128-byte READ request	6'b110111		

Table 3. 15: Request Commands

3.6.3.2 Response Commands

All response commands are issued by the responder to the requester.

Symbol	Description	CMD Bit	LNG
RD_RS	READ response	6'b111000	1 + Data Flits
WR_RS	WRITE response	6'b111001	1
ERROR	ERROR response	6'b111110	1

Table 3. 16: Response Commands

3.6.3.3 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.

Symbol	Description	CMD Bit	LNG
Null	Null	6'b000000	NA
PRET	Retry pointer return	6'b000001	1
TRET	Token return	6'b000010	1
IRTRY	Init retry	6'b000011	1

Table 3. 17: Flow Commands

3.7 Error detection

Error detection in HMC is an important aspect of ensuring reliable operation of the memory system. There are serval techniques used for error detection in HMC memory , including:

1. Error Correction Code (ECC): ECC is a technique that adds extra bits to the memory data to detect and correct .HMC uses various types of ECC such as single-bit error correction and double-bit error detection codes to detect and correct errors in memory.
2. Parity Checking : parity checking is a technique that adds an additional bit to the memory data to detect errors . in the technique, the total number of ones is odd or even , it indicates that an error bas occurred.
3. Cyclic redundancy check (CRC) : CRC is a technique widely used in communication systems to detect errors . in HMC , CRC is used to detect errors in HMC in this technique , a checksum is generated for the data and is compared with the received checksum. If there is a mismatch , it indicates an error.
4. Scrubbing : scrubbing is a technique used to detect and correct errors by periodically reading and rewriting data in memory. The technique prevents errors from accumulating in memory over time.

There are many ways to detect errors, but HMC uses ERROR DETECTION using CRC as it detects it using checksum and we described it in section 3.4.2

Chapter 4: Verification Plan and Simple Test Case

4.1 Verification plan

In this section we will discuss the verification plan and design specs we followed to build our verification environment. Before verifying any system, it is easier to set our minds and organize it before implementing the verification environment and that is done in two steps, the first step is to extract the specifications of the design and the second step is to use the extracted specs to write a verification plan which makes the verification cycle easier.

4.1.1 Introduction about Verification plan

Any Design project has specification files. Both the design and the verification engineer have to use these specification files into two different ways, the design engineer has to implement the design according to the project specs and the verification engineer has to check that the designs follow the specs as well. The role of verification engineer is in parallel to the design engineer. The verification engineer should extract the main points that will be used through the verification cycle, such as:

- Interfaces and their main signals.
- How to drive those signals.
- Main system protocol and modes of operations.
- What can be checked.
- Which points should be covered to say that this project is verified.
- Suggested test scenarios.

4.1.2 Verification plan Tabs

- *Interface*: it describes the signals in the interfaces and how they operate.
- *Driving pin level*: it describes how we can generate the signals to be fed through the interfaces.
- *Generation:* it describes the generation of signals in some situations, such as the sequence that will be generated during the initialization mode.
- *System:* it has a summary of modes of operation and generation for some main flow signals.
- *RF (Register file):* it describes the addresses of the register file and its registers.
- *Checkers:* it has all the interface signals and how we can check the device by driving those signals
- *Coverage:* has all signals needed to be covered.
- *Test scenarios:* Suggested test sequences, “Must” tests and “Should” tests.

Figure 4. 1: Verification plan sheet tabs

Parameter	Default Size
FPW	2
NUM_LANES	16
LOG_MAX_RX_TOKENS	8
LOG_MAX_HMC_TOKENS	10
HMC_RX_AC_COUPLED	1
DETECT_LANE_POLARITY	1
CTRL_LANE_REVERSAL	1
CTRL_SCRAMBLERS	1
OPEN_RSP_MODE	0
RX_RELAX_INIT_TIMING	1
RX_BIT_SLIP_CNT_LOG	5
SYNC_AXI4_IF	0

Table 4. 1: Configuration Parameters

4.1.2.1 Test Scenarios

Basically, here we typed our suggested test cases for testing the HMC controller with them. So, we classify those tests into 2 types: The first type is: “Must tests”, those tests must work properly, and they consist of the design main modes of operations and transactions between them, and the design main features that the design should make them in normal. The second type is: “Should tests”, if we don’t make those tests nothing will happen, because those tests include additional and not necessary features.

Test Scenarios	Priority
Design Must do initialization protocol well.	must
Design Must be able to go to sleep mode.	must
Design Must be able to go to IDLE mode.	must
Design Must be able to go to HMC Retry mode.	must
Design Must be able to go to TX Retry mode.	must
Design Must be able to transact from IDLE to sleep mode	must
Design Must be able to transact from HMC Retry mode to TX Retry mode	must
Design Must be able to transact from sleep mode to IDLE mode throw initialization mode	must
Design Must be able to do HMC reset protocol.	must
Design Must be able to do user reset protocol.	must
Design Must be able to deal with user-clk and hmc-clk if they are from different sources.	must
Design Must be able to do Read operation	must
Design Must be able to do Write operation	must
Design Must be able to check if Read-enable and Write-enable both are set.	must

Table 4. 2: Must Tests

Test Scenarios	Priority
Design Should check that the size of each TX output lane are less than 85 bits.	should
Design Should check TVALID and TReady during giving data through the AXI-4 master.	should
Design Should check TVALID and TReady during taking data through the AXI-4 slave.	should
Design Should check Error abort flag during the HMC retry mode or TX retry mode.	should
Design Should if the TX of the controller gives start_retry packets to the HMC during the HMC retry operation.	should
Design Should if the TX of the HMC gives start_retry packets to the HMC controller during the TX retry operation.	Should

Table 4. 3: Should Tests

4.1.2.2 Checkers

Here we have our main checks, we are checking on some main fields in the request and response packets.

Design Requirement Description		Checking Requirement Description
Request Packet Header [63 : 0]	Addr : Request address. For some commands, control fields are included within this range. [57: 24]	Check HMC configurations whether it is 4GB or 8GB and Max Block Size configuration whether it is 32 , 64 , 128 , 256 bits. Then finally check if the given address meets the form of the specified address form for the applied configurations.
	TAG : [22: 12] 1) TAG is uniquely identified for each packet 2) remains associated with the request until a response is received indicating that the request has been executed 3) There is no TAG field in posted requests, like post write, as it doesn't have response 4) 11 bits long, which can store up to 2048 TAG	Check that it is unique for each packet, Check that there is no TAG in posted requests, Check that the TAG assigned with the request is excluded from a response to be able to be used again. Check if it is excluded and it is NULL, TRET or posted packet or not. Check the case when the TAG is equal to CUB.
	LNG : [11: 7] : indicates the total number of FLITs in the packet.	Check and compare the LNG of the packet header with the token count register to know this packet will be transmitted or not.
	CMD : [6: 0] Request Commands indicate the packet type	Checking each command we sent, and if it is posted or not. Checking the encoding correctness of commands. Check the that the LNG is identical to the required size in the command.
Request packet between HMC and HMC Controller, the controller will set some values in the tail of each packet.		1. Check CRC calculated by the HMC controller. 2. Compare between the request packet at the Axi Slave TX interface and the Request packet at the Transceiver interface. "Command, tag and packet length"

Table 4. 4: Checks in Request packet

Design Requirement Description		Checking Requirement Description
Response Packet Header [63 : 0]	AF : Atomic flag [33]	Check that this is set when the CMD contains the code of an atomic command.
	TAG : [22: 12] 1) TAG is uniquely identified for each packet 2) remains associated with the request until a response is received indicating that the request has been executed 3) There is no TAG field in posted requests, like post write, as it doesn't have response 4) 11 bits long, which can store up to 2048 TAG	Check that it is unique for each packet, Check that there is no TAG in posted requests, Check that the TAG assigned with the request is excluded from a response to be able to be used again. Check if it is excluded and it is NULL, TRET or posted packet or not. Check the case when the TAG is equal to CUB.
	LNG : [11: 7]: indicates the total number of FLITs in the packet.	Check and compare the LNG of the packet header with the token count register to know if this packet will be transmitted or not.
	CMD : [6:0] Response Commands indicate the packet type	Checking each command we sent, and if it is posted or not. Check the that the LNG is identical to the required size in the command.
Response Packet Tail [63 : 0]	CRC : error-detecting code field [63: 32]	Check whether this CRC is flipped, and this is a poisoned packet or not.
	RTC : Return token count for transaction-layer flow control. [31:29]	Check that RTC is increased in the controller TX when there is no traffic by sending TRET packets. Check that the token -of each valid FLIT is stored in controller RX input buffer or transmitted from the controller input buffer- is returned to the controller TX.
	FRP : Retry pointer representing this packet's position in the retry buffer [17:9]	Check that it is unique for each FLIT transmitted. Check that it is returned to the controller TX after the FLIT went inside the controller RX input buffer or being transmitted to outer agents, to be used again. Check this pointer in HMC retry mode and TX retry mode.
	RRP : Retry pointer being returned for the other side of link [8: 0]	Check that it is unique for each FLIT transmitted. Check that it is returned to the controller TX after the FLIT went inside the controller RX input buffer or being transmitted to outer agents, to be used again. Check this pointer in HMC retry mode and TX retry mode.
Response packet between HMC and HMC Controller, the Reactive agent will generate it according to the request of the user.		1. Check the tail of the packet "CRC not changed". 2. Compare between the request packet at the AXI Slave TX interface and the Request packet at the Transceiver interface. "command, tag and packet length"

Table 4. 5: Checks in Response Packet

4.1.2.3 Coverage and Assertions

4.1.2.3.1 Coverage

ID		Design Requirement Description	Coverage Requirement Description
DR14	Request Packet Header [63 : 0]	LNG : [11: 7] 1) indicates the total number of FLITs in the packet. 2) A packet that contains no data FLITs would have LNG = 1	1. cover packet lengths "number of flits in a packet" [1:9].
DR15		DLN:[14:11] Duplicate of packet length field.	1. cover Duplicate of packet length "number of flits in a packet" [1:9].
DR16		CMD : [6: 0] Request Commands indicate the packet type 1. WRITE Requests 2. POSTED WRITE Requests 3. READ Requests 4. ARITHMETIC ATOMICS 5. BOOLEAN ATOMICS 6. COMPARISON ATOMICS 7. BITWISE ATOMICS	1. cover the write and mode write modes: HMC_WRITE_16, HMC_WRITE_32, HMC_WRITE_48,HMC_WRITE_64, HMC_WRITE_80,HMC_WRITE_96, HMC_WRITE_112,HMC_WRITE_128 2. cover the posted write with different sizes "flits num" : HMC_POSTED_WRITE_16, HMC_POSTED_WRITE_32, HMC_POSTED_WRITE_48, HMC_POSTED_WRITE_64, HMC_POSTED_WRITE_80, HMC_POSTED_WRITE_96, HMC_POSTED_WRITE_112, HMC_POSTED_WRITE_128 3. Cover ATOMIC_Commands: Dual 8-byte add, Single 16-byte add 4. cover the READ Request : HMC_READ_16, HMC_ READ _32, HMC_ READ _48,HMC_ READ _64, HMC_ READ _80,HMC_WRITE_96, HMC_ READ _112,HMC_ READ_128, mode_read 5. Cover POSTED ATOMIC_Commands: POSTED Dual 8-byte add, POSTED Single 16-byte add 6. Cover consecutive write commands with the same size. 7.

			Cover consecutive posted write commands with the same size. 8. Cover consecutive read commands with the same size. 9.cover CMD_transitions: -write cmd then read cmd then write cmd, -posted write cmd then read cmd then posted write cmd, -read cmd then write cmd then read cmd, -posted write cmd then read cmd then write cmd,

Table 4. 6: Request packet Coverage in verification plan:

ID	Design Requirement Description		Coverage Requirement Description
DR17	Response Packet Header [63 : 0]	LNG : [11: 7] 1) indicates the total number of FLITs in the packet. 2) A packet that contains no data FLITs would have LNG = 1	1. cover packet lengths "number of flits in a packet" [1:9].
DR18		DLN:[14:11] Duplicate of packet length field.	1. cover Duplicate of packet length "number of flits in a packet" [1:9].
DR19		CMD : [6: 0] Response Commands indicate the packet type 1. READ_response 2. WRITE_response 3. ERROR_response 4.MODE_READ_response 5. MODE_WRITE_response	1. cover the response command: READ_response, WRITE_response, ERROR_response,MODE_READ_response MODE_WRITE_response 2.cover flow commands: Null, Retry_pointer_return, Token_return, Init_retry.

Table 4. 7: Response packet Coverage in verification plan

4.1.2.3.2 Assertions

The behavior of a system can be written as an assertion that should be always true. Hence assertions are used to validate the behavior of a system defined as properties and can also be used in functional coverage.

- **Register File Assertions**

  Assertions were used to check timing and different signals on command pins there are 6 assertions with all of them passing without any miss. These assertions were implemented inside the RF interface and all of them not working in the reset state.

Assertion	Description
wen_valid	wen on when address and write data not x or z.
ren_valid	ren set when address not x or z.
wen_ren_not_together	wen and ren not set on the same time.
access_complete_valid	access_complete not set after triggering wen or ren from 1 to 0.
access_complete_1_cycle	access_complete not take only one cycle and be 0.
invalid_addr_access_complete_together	access_complete and invalid_address not at the same cycle

Table 4. 8: RF assertions

```
** Info: ASS1:Assertion passed, Write enable valid.
   Time: 50 ns Started: 50 ns  Scope: HMC_top.rf_if.ass1 File: C:/Users/HP-USER/Desktop/For_initialization_test/RF_IF.sv Line: 53

** Info: ASS2:Assertion passed, read enable valid.
   Time: 100 ns Started: 100 ns  Scope: HMC_top.rf_if.ass1 File: C:/Users/HP-USER/Desktop/For_initialization_test/RF_IF.sv Line: 58

** Info: ASS3:Assertion passed, Write and read enable not on the same clk cycle.
   Time: 50 ns Started: 50 ns  Scope: HMC_top.rf_if.ass3 File: C:/Users/HP-USER/Desktop/For_initialization_test/RF_IF.sv Line: 63

** Info: ASS4:Assertion passed, Write enable valid.
   Time: 60 ns Started: 50 ns  Scope: HMC_top.rf_if.ass4 File: C:/Users/HP-USER/Desktop/For_initialization_test/RF_IF.sv Line: 68

** Info: ASS5:Assertion passed,  access complete valid.
   Time: 120 ns Started: 110 ns  Scope: HMC_top.rf_if.ass5 File: C:/Users/HP-USER/Desktop/For_initialization_test/RF_IF.sv Line: 73

** Info: ASS6:Assertion passed, access complete and invalid address not together.
   Time: 230 ns Started: 220 ns  Scope: HMC_top.rf_if.ass5 File: C:/Users/HP-USER/Desktop/For_initialization_test/RF_IF.sv Line: 78
```

Figure 4. 2: Assertions

4.2 Simple Test Case

In this section we will discuss how the initialization is done and explain two test cases, one for writing and the other for reading the written packet.

4.2.1 Power-on and Initialization of the HMC-Controller

According to the documentation, the initialization is done by the following steps:

1. Apply clk_hmc and hold hmc_res_n low.
2. Set hmc_res_n = 1.
3. I2C is used to load the internal HMC registers.
4. Set hmc_init_cont_set when Register Load has finished.
5. openHMC controller performs initialization automatically.

But these steps can be used only when the HMC Memory is connected to the HMC Controller, so we have to do the initialization steps manually in the agent that is acting as the HMC

memory, we must be very careful when the TX and RX moves from case to case to drive the proper data on the Link between the controller and the memory agent. The following FSM is used to perform the initialization and is implemented in the Hmc_Mem_Driver.

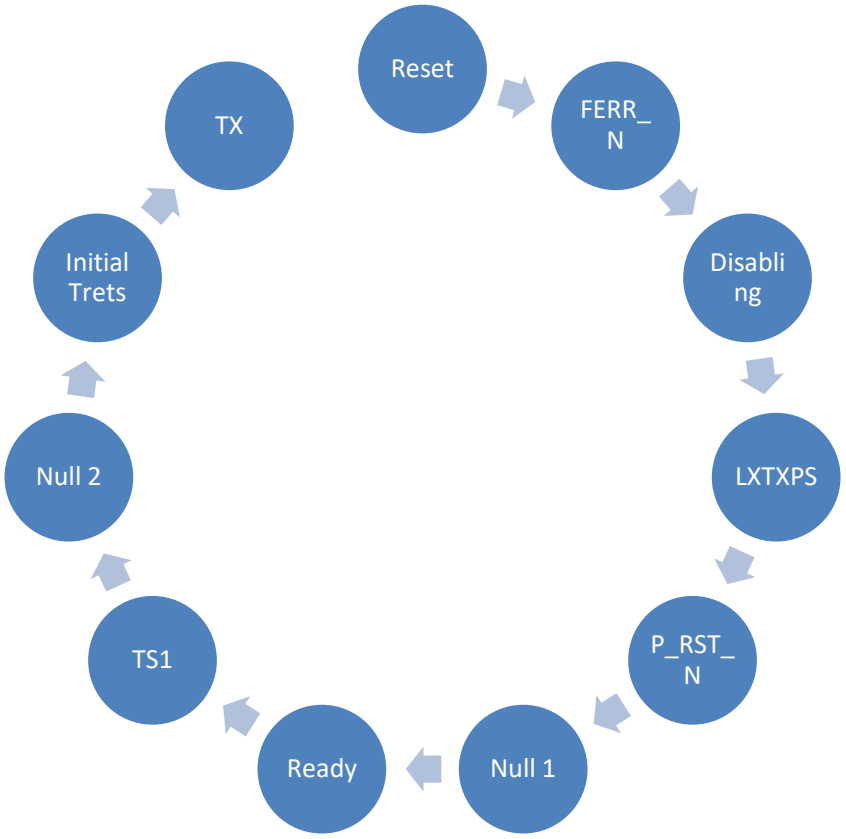


Figure 4. 3: RX FSM

1. *Reset:* Apply clk_hmc and clk_user and hold hmc_res_n and user_res_n low (active low resets), and after 3 clk cycles set the two resets to 0.
2. *FERR_N:* After 1 clk cycle, set FERR_N=1, FERR_N: Fatal error indicator. it drives LOW if fatal error occurs.
3. *Disabling:* Disable scrambler and run length limiter using the register file sequence (RF_Disable_scrambler_Sequence).
4. *LXTXPS:* After 4 clk cycle, set LXTXPS to 1.
5. *P_RST_N:* Set P_RST_N using the register file sequence (RF_Set_P_RST_N_Sequence).
6. *Null 1:* After 3 clk cycles, set phy_data_rx_phy2link to 0.
7. *Ready:* Set phy_tx_ready, phy_rx_ready signals. Set the init_cont_set using the register file sequence(RF_Set_init_cont_Sequence).
8. *TS1:* Set specific training sequences and wait until all descramblers are aligned or synchronized. Figure 4.4 shows a sample of the code used to generate the TS1.

```
@(posedge hmc_mem_if.hmc_clk);
for(bit[15:0] i=0; i<=15; i++) begin
    hmc_mem_if.phy_data_rx_phy2link =
       {{16'hF0C0 + i},{14{16'hF050 +i}},{16'hF030 + i}};
    wait(hmc_mem_if.phy_bit_slip !=0);
    wait(hmc_mem_if.phy_bit_slip ==0);
    repeat(2)@(posedge hmc_mem_if.hmc_clk);
    wait(hmc_mem_if.phy_bit_slip ==0);
    if(hmc_mem_if.phy_bit_slip ==0)begin
        wait(hmc_mem_if.phy_bit_slip !=0);
        wait(hmc_mem_if.phy_bit_slip ==0);
        wait(hmc_mem_if.phy_bit_slip !=0);
    end
    else if (hmc_mem_if.phy_bit_slip !=0) begin
         wait(hmc_mem_if.phy_bit_slip ==0);
         wait(hmc_mem_if.phy_bit_slip !=0);
    end
end
```

Figure 4. 4: TS1 Generation

9. Null 2: Send null packet, after this operation the Link Up in Status General RF is set which means Link is ready for operation.
10. Initial Trets: Send one or more (token return packet) on (phy_data_rx_phy2link input).
11. TX: In this state the packets are sent.

4.2.2 Write on the HMC Storage

To enable writing on the HMC_Mem_Storage the AXI_Req_Agent should send a write request packet with tail fields equal 0 and the command in the header field is a write command. Through the AXI_Req_Driver drives the valid signal, TDATA and TUSER on the AXI_Req_IF. The TX of the controller collects the packet, fills its tail, and sends it to the phy_data_tx_link2phy, The HMC_Mem_Monitor collects the packet, converts it to transaction level and sends it to the HMC_Mem_Storage. The HMC_Mem_Storage takes this packet (Request Packet) and if the CMD is a write command, it will store this packet in an associative array, the packet address is the address that it was in the header of this packet. Finally, it sends a Response packet to the HMC_Mem_Driver which indicates that the process is done. The fields of this packet are the same as the Request packet, but the CRC field is updated. The HMC_Mem_Driver converts this packet to pin level and drives it to the HMC Controller. The

HMC Controller calculates the CRC once again and compares it with the CRC in the Response packet. If it is true, the packet passes to the AXI_Rsp_Agent side. The AXI_Rsp_Agent must set the TREADY signal, so that it can receive the Response packet through the TDATA and convert it to transaction level. Finally, the Scoreboard compares the two Request packets and the two Response packets on both sides.

4.2.3 Read from the HMC Storage

To enable reading from the HMC_Mem_Storage the AXI_Req_Agent should send a read request packet with tail fields equal 0 and the command in the header field is a read command. The address of the Request packet must be used once before while sending a write Request packet. Through the AXI_Req_Driver drives the valid signal, TDATA and TUSER on the AXI_Req_IF. The TX of the controller collects the packet, fills its tail, and sends it to the phy_data_tx_link2phy, The HMC_Mem_Monitor collects the packet, converts it to transaction level and sends it to the HMC_Mem_Storage. The HMC_Mem_Storage takes this packet (Request Packet) and if the CMD is a read command, it will read the request packet that was stored in the storage and generate a Response packet contain the data that was stored. The HMC_Mem_Storage also calculates the CRC of this packet and fills the CRC field and updates the command field of this packet. . Finally, it sends a Response packet to the HMC_Mem_Driver. The HMC_Mem_Driver converts this packet to pin level and drives it to the HMC Controller. The HMC Controller calculates the CRC once again and compares it with the CRC in the Response packet. If it is true, the packet passes to the AXI_Rsp_Agent side. The AXI_Rsp_Agent must set the TREADY signal, so that it can receive the Response packet through the TDATA and convert it to transaction level.

4.3 Modes of Operation

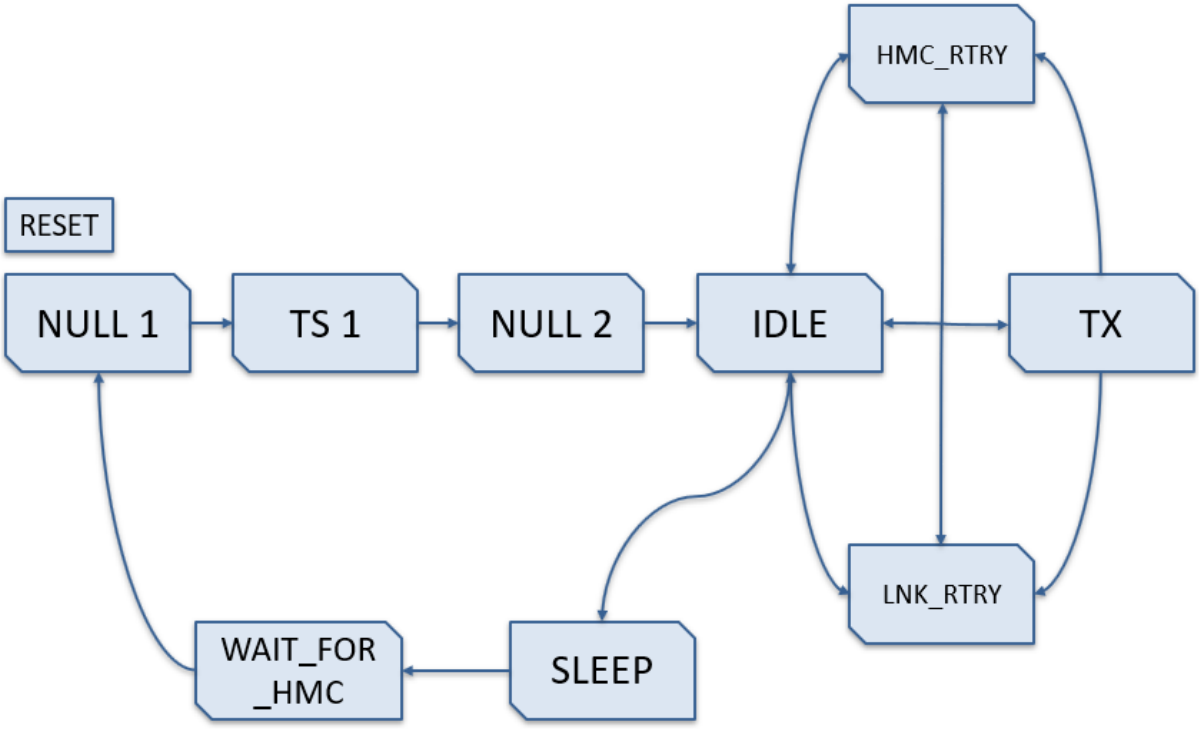


Figure 4. 5: TX FSM

4.3.1 TX Mode

The normal mode in which the packets “coming from the AXI request interface” flow from the controller transmitter to the HMC receiver, and then the HMC make the suitable response and send it back through its transmitter, then the controller receiver receives the response from the HMC and does its checks, finally the packet went out through the AXI response interface.

4.3.2 Sleep Mode and Down Mode

It is a mode in which the HMCC goes into power saving mode or down mode, to reduce power usage, and stop the controller and the memory cube from sending and receiving normal packets transactions.

Each link can independently be set into a lower power state through the usage of the power state management pins, LxRXPS and LxTXPS.

Each of the links can be set into a low-power state, sleep mode, as well as a minimum power state called down mode

Sleep Mode:

1- To enter sleep mode:
   a. Set set_hmc_sleep =1 "control" : sleep_mode "status" , Transition of the LxTXPS from 1 to 0.
   b. HMC will acknowledge going into sleep mode by:
      Transition of the LxRXPS from 1 to 0, LINKPOWERMODE = 0 (just sleep mode)
2- Exit sleep mode:
   a. Set set_hmc_sleep = 0 : sleep_mode status register
   b. Transition of the LxTXPS from 0 to 1.
   c. HMC will acknowledge exiting sleep mode by:
      Transition of the LxRXPS from 0 to 1.

Down Mode:

To enter Down Mode, the only difference is that the memory cube will acknowledge going into Down-Mode by 2 actions:

1- LINKPOWERMODE = 1
2- Transition of the LxRXPS from 1 to 0.

4.3.3 TX Link Retry Mode

In case of an error on the TX path from requester to responder, the HMC will request a link retry.

In other words, when the memory cube receives a packet from the controller transmitter, it starts to check on 3 things: (Sequence number, CRC, and LNG).

If any one of those checks gives error, the memory cube will request TX Link Retry in which the cube sends some requests to the controller asking it to retransmit the last packet which has error.

Note: They can identify the required packet to get retransmitted by the packet sequence number.

TX links retry mechanism steps:

1- The HMC detects an error in the received packet.
2- The HMC issues a programmable series of start_retry packets to the RX link to force a link retry.
   Start_retry packets have the 'StartRetryFlag' set (FRP[0]=1).
3- When the irtry_received_threshold at the Receive (RX)-Link is reached, the Transmit (TX)

link starts to transmit a series of clear_error packets that have the 'ClearErrorFlag' set (FRP[1]=1).

4- Afterwards, the TX link uses the last received RRP as the RAM read address and re-transmits any valid FLITs in the retry buffer until the read address equals the write address, meaning that all pending packets were re-transmitted.
5- Upon completion the RAM read address returns to the last received RRP.
6- Re-transmitted packets may therefore be re-transmitted again if another error occurs.

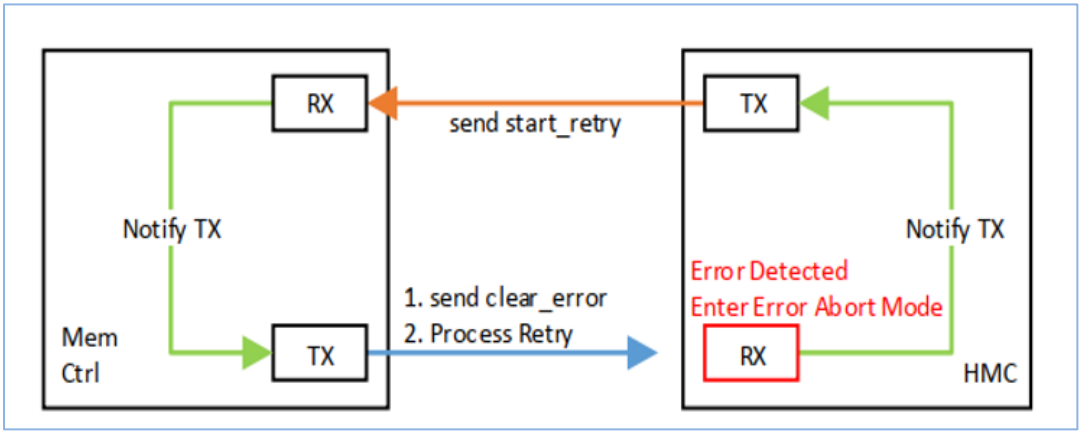


Figure 4. 6: TX Link Retry

4.3.4 HMC Retry Mode:

In case of an error on the RX path from responder to requester, the RX link will request a link retry.

In other words, when the receiver of the controller receives a response packet from the memory cube, it will do some checks like (CRC, RRT, FRT, scrambler, sequence item, and so on "as described before in controller RX link Figure## {point to figure of controller RX link}").

If any one of those checks discovered an error in the received packet, the controller will send packets requesting the controller transmitter to retransmit this packet again.

RX HMC Retry mechanism steps:

1- The controller receiver detects an error in the received packet.
2- The TX link will send start_retry packets where upon the responder will start to re-transmit all packets that were not acknowledged by the RRP yet.
3- Meanwhile, the RX link remains in the so-called error_abort_mode where all subsequently incoming packets are
dropped.

4- The TX link monitors this state and sends another series of start_retry packets if the error_abort_mode was not cleared after 250cycles.

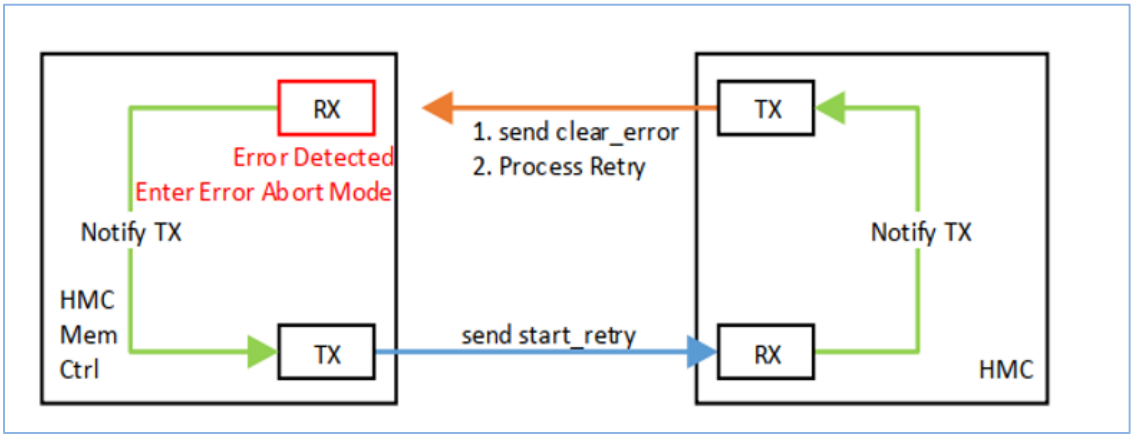


Figure 4. 7: HMC Retry

Chapter 5: Environment Implementation

5.1 UVM Environment

UVM is a methodology for functional verification using SystemVerilog, complete with a supporting library of SystemVerilog code. The letters UVM stands for the Universal Verification Methodology. UVM was created by Accellera based on the OVM (Open Verification Methodology) version 2.1.1. The roots of these methodologies lie in the application of the languages IEEE 1800™ SystemVerilog, IEEE 1666™ SystemC, and IEEE 1647™ e. UVM is a methodology for the functional verification of digital hardware, primarily using simulation. The hardware or system to be verified would typically be described using Verilog, SystemVerilog, VHDL or SystemC at any appropriate abstraction level.

5.1.1 Introduction

UVM testbenches are complete verification environments composed of reusable verification components and used as part of an overarching methodology of constrained random and coverage-driven verification. The UVM Class Library provides generic utilities, such as component hierarchy, transaction library model (TLM), configuration database, etc., which enable the user to create virtually any structure wanted for the testbench. The UVM class library makes a clear separation between the sequence which generates stimulus and the structure which constructs verification environment.
In addition to a rich base class library, it provides a reference best-practice verification methodology. Fully supported by major tool vendors and maintained by an industry recognized body (Accellera).

5.1.2 Advantages of UVM Based Testbench

- UVM methodology provides scalable, reusable, and interoperable testbench development.
- To have uniformity in the testbench structure across the verification team, UVM provides guidelines for testbench development.
- UVM provides base class libraries so that users can inherit them to use inbuilt functionality.
- The driver-sequencer communication mechanism is an inbuilt mechanism in UVM that reduces verification efforts for the connection.
- UVM also provides verbosity to control message displays.

5.1.3 UVM Class Hierarchy

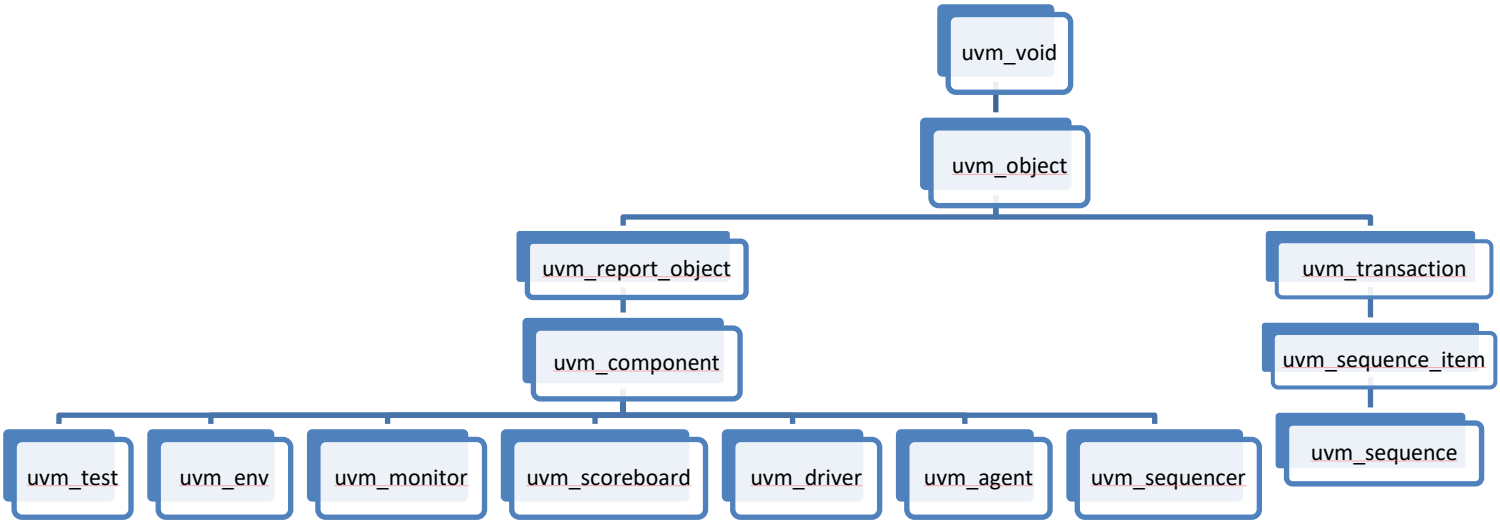


Figure 5. 1: UVM Class Hierarchy

5.1.4 UVM Phases

UVM Phases are a synchronizing mechanism for the environment. Phases are represented by callback methods; A set of predefined phases and corresponding callbacks are provided in uvm_component. The Method can be either a function or task. Any class deriving from uvm_component may implement any or all these callbacks, which are executed in a particular order. The UVM Phases are shown in Figure 4.2.

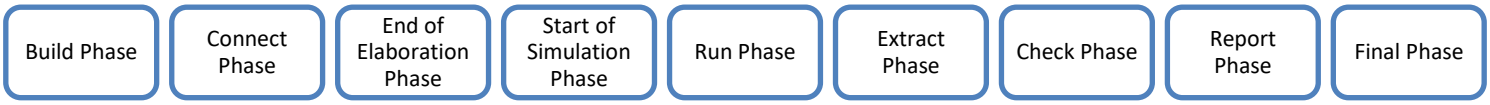


Figure 5. 2: UVM Phases

Build Phase is used to instantiate sub-components, instantiate register model, get configuration values for components being built and set configuration values for sub-components. *Connect phase* is used to connect TLM ports and exports of all components. *End of elaboration phase* is used to make any final adjustments to the structure, configuration, or connectivity of the testbench before simulation starts. *Start of simulation phase* is used for printing configuration information. *Run phase* is used for stimulus generation, driving, monitoring, and checking. *Extract phase* is used to retrieve and process information from scoreboards and functional coverage monitors. *Check phase* is used to check that the DUT behaved correctly and to identify any errors that may have occurred during the execution of the test bench. *Report phase* is used to display the results of the simulation or to write the results to file. And finally*, the Final phase* is used to complete any other outstanding actions that the test bench has not already completed.

5.2 System Architecture Diagram of The HMC Controller

System architecture diagrams provide a visual illustration of a system's various components and show how they communicate and interact with each other.

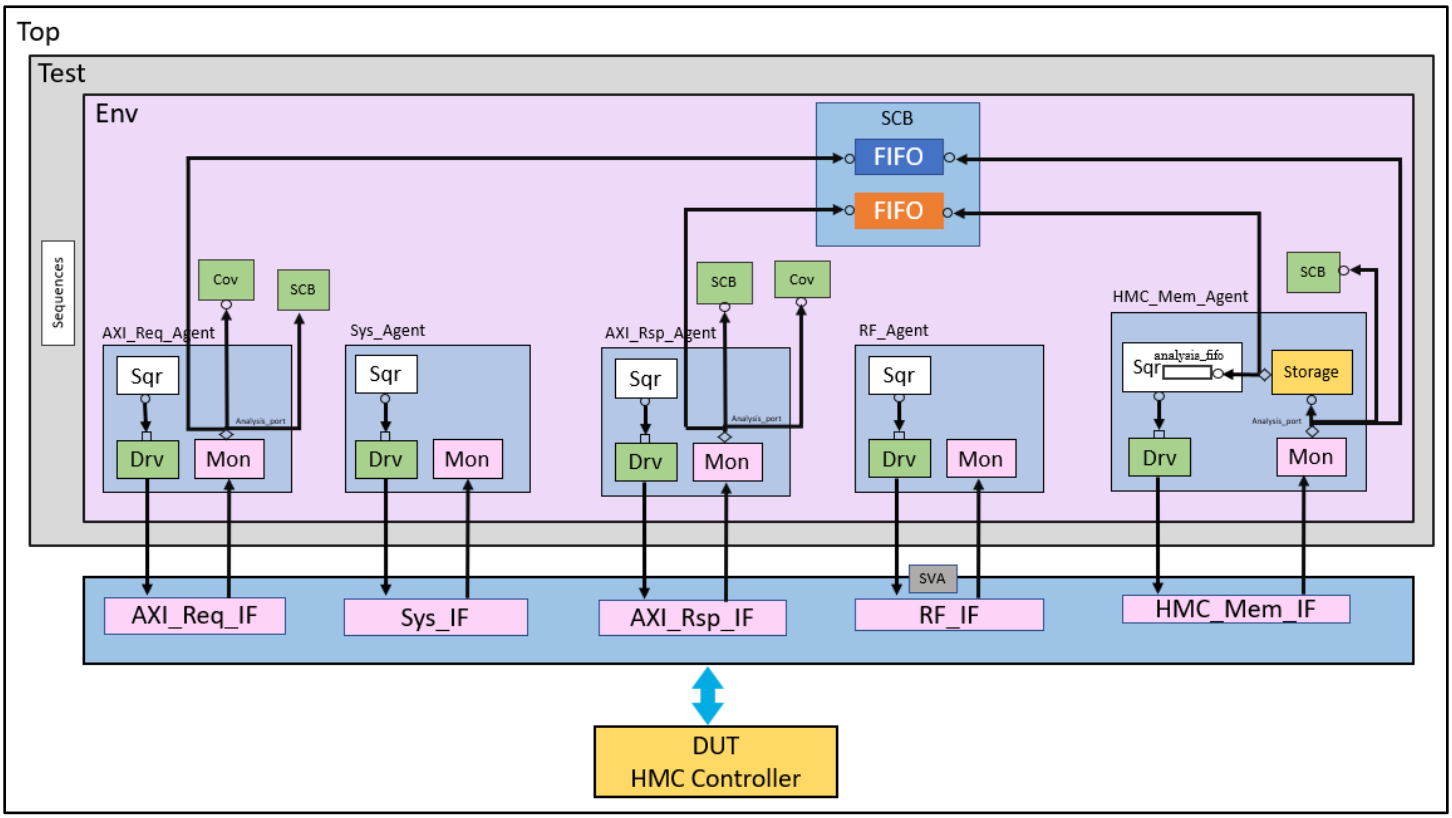


Figure 5. 3: System Architecture Diagram of The HMC Controller

5.2.1 The Main Function of Each UVM Components

1. *Sequencer:* it is a mediator who establishes a connection between sequence and driver, it passes transactions or sequence items to the driver so that they can be driven to the DUT.
2. *Driver:* it interacts with DUT. It drives randomized transactions or sequence items to DUT as a pin-level activity using an interface. The driver has to be extended from uvm_component names as uvm_driver. The transaction or sequence items are retrieved from the sequencer and the driver drives them to the design using the interface handle. An interface handle can be retrieved from the configuration database which was already set in the top-level hierarchy.
3. *Monitor:* it is a passive component used to capture DUT signals using a virtual interface and translate them into a sequence item format. These sequence items or transactions are broadcasted to other components like the UVM scoreboard, coverage collector, etc. It uses a TLM analysis port to broadcast transactions. It has to be extended from uvm_monitor which is derived from uvm_component.

4. *Agent:* it encapsulates a Sequencer, Driver and Monitor into a single entity by instantiating and connecting the components together via TLM interfaces, since UVM is all about configurability an agent can also have configuration options like the type of UVM agent active or passive.
5. *Scoreboard:* it is a verification component that contains checkers and verifies the functionality of the design. It receives transactions from the monitor using the analysis export for checking purposes.
6. *Environment:* it provides a well-mannered hierarchy and container for agents, scoreboards, coverage collectors and other verification components including other environment classes that are helpful in reusing block-level environment components at the SoC level. The user-defined env class has to be extended from the uvm_env class.
7. *Test:* it is at the top of the hierarchical component that initiates the environment component construction. It is also responsible for the testbench configuration and stimulus generation process. Based on the features and functionalities of the design mentioned in the verification plan, tests are written. The user-defined test class is derived from uvm_test.

5.2.2 The Main Function of Each UVM Object

1. *Sequence item:* extended from the uvm_sequence_item class as it leverages generating stimulus and has control capabilities for the sequence-sequencer mechanism. As shown in figure 4.1 uvm_sequence_item is derived from the uvm_transaction class. It supports all methods like copy, compare, clone, print, etc.
2. *Sequence:* is a container that holds data items (uvm_sequence_items) which are sent to the driver via the sequencer. uvm_sequence class is parameterized with uvm_sequence_item.

Let's take a bottom-up approach while building the UVM environment starting from the HMC Controller interfaces till we have a complete environment.

5.2.3 HMC Controller interfaces

There are five interfaces to communicate with the RTL design: System interface, Register File interface, AXI Request interface, HMC Memory interface and AXI Response interface discussed in Section 3.5 HMC Controller interfaces. These five interfaces will also be used here in the UVM environment to drive and monitor the HMC Controller.

5.2.4 Sequence items

They are the data members which stimulate the DUT ports. They Pass the values to the DUT ports and receive the response values from the DUT ports. Since there are five different interfaces, we will need six different types of Sequence item: Sys_Sequence_Item for System Interface, RF_Sequence_Item for Register File Interface, AXI_Req_Sequence_item for AXI Request Interface, AXI_Rsp_Sequence_item

for AXI Response Interface, and HMC_Req_Sequence_item and HMC_Rsp_Sequence_item for the HMC memory Interface , they are used to handle data between agents and interfaces.

5.2.5 Agents

As we have five interfaces, we need five agents to communicate with the HMC Controller. There are three types of agents: Active agent, Passive agent, and Reactive agent. *Active Agent* generates the sequences and drives it to the Dut through an interface and includes Sequencer, Driver, and Monitor. *Passive Agent* doesn't drive stimulus to the DUT, it instantiates only a monitor component which means it includes a Monitor only. *Reactive Agent* waits for a request from DUT and according to this request it will generate a response and drive it to the Dut, the agent types are shown in Figure 4.4.

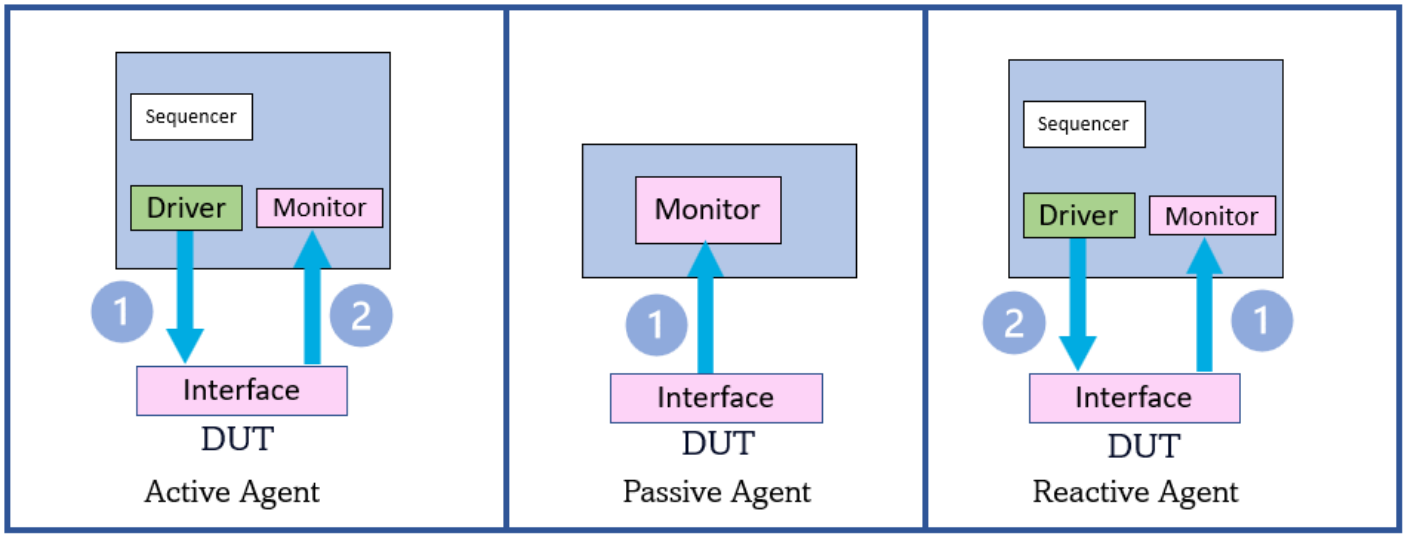


Figure 5. 4: Agent Types

In our UVM environment we will build five agents one is Reactive agent and will call it HMC_Mem_Agent and the remaining four agents are Active agents and will call them Sys_Agent, RF_Agent, AXI_Req_Agent and AXI_Rsp_Agent.

5.2.5.1 System Agent (Sys_Agent)

This Agent is Responsible for generating the two clocks and Resets of the system, the two resets are synchronized to its clocks. according to the design the two clocks can be derived from the same source or from different sources and this depends on SYNC_AXI4_IF parameter, if it was set to 0, both clocks would be from the same source and it was set to 1, the two clocks would be from two different sources. The System Interface was mentioned in section 3.5.1.

In this section we will talk about all components of this agent in brief.

5.2.5.1.1 System Sequence item (Sys_Sequence_Item)

It was extended from uvm_sequence_item, uvm_sequence_item inherits from the uvm_object via the uvm_transaction class. therefore uvm_sequence_item is of an object type.

It is a data class, defined to hold random input clk_user, clk_hmc, res_n_user and res_n_hmc and will be randomized inside the sequence before it is sent to the driver. The class has utility macros, these macros provide implementations of the create method needed for copy, clone, print and the get_type_name method and these methods are needed for debugging. Also, it has the new constructor that is used for initializing the class properties.

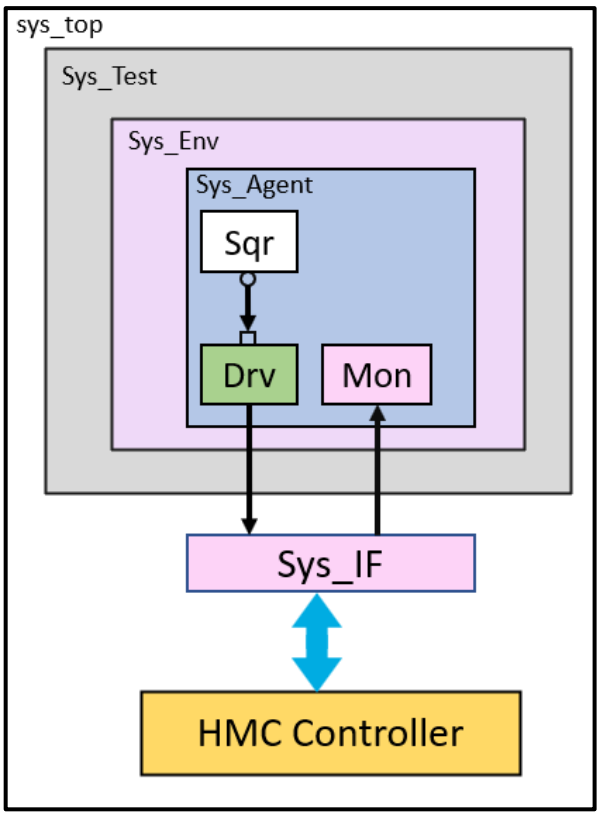


Figure 5. 5: System Agent in UVM Env

5.2.5.1.2 System Driver (Sys_Driver)

This class was extended from uvm_driver and registered it in the factory, then we declared virtual interface handle to retrieve actual interface handle using configuration database in the build_phase. We write standard new() function since the driver is a uvm_component. The new() function has two arguments as string name and uvm_component parent. We Implemented the build_phase and get interface handle from the configuration database. Finally, we Implemented the run_phase to get the sequence items and drive them to the DUT using a virtual interface handle. Inside the run_phase there is a task called drive() within it we implement how the 2 clocks will be generated; the period of both clocks is 10ns, so it toggles each 5ns. The clocks are 50% duty cycle also the 2 resets are generated as we know the reset signals are active low and are synchronized so the two resets will be fixed to 0 within 30ns from the beginning then they will 1 to the end of simulation.

5.2.5.1.3 System Monitor (Sys_Monitor)

This class is designed to capture signal level information and translate it into the transaction. This class was extended from uvm_monitor and register it in the factory, then we declared virtual interface handle to retrieve actual interface handle using configuration database in the

build_phase. Then we declared an analysis port to broadcast the sequence items or transactions but was commented because there isn't any component listening to it. We write standard new() function. Since the monitor is a uvm_component. The new() function has two arguments as string name and uvm_component parent. We Implemented the build_phase and get interface handle from the configuration database. Finally, we Implemented the run_phase to sample DUT interface using a virtual interface handle and translate into transactions. The write() method sends transactions to the collector component.

5.2.5.1.4 System Sequencer (Sys_Sequencer)

It passes the sequence item to the driver so that they can be driven to the DUT. This class was extended from the parameterized base class "uvm_sequencer" which is parameterized by request item type, and we registered it in the factory. We write standard new() function. Since the sequencer is an uvm_component. The new() function has two arguments as string name and uvm_component parent.

5.2.5.1.5 System Agent (Sys_Agent)

This class was extended from uvm_agent and registered it in the factory. In the build_phase, we instantiated driver, monitor, and sequencer because it is an active agent. In the connect_phase, connect driver and sequencer components.

5.2.5.1.6 System Environment (Sys_Env)

This class is a container for agents, scoreboards, and other verification components. This class was extended from uvm_env and registered in the factory. In the build_phase, we instantiate the agent.

5.2.5.1.7 System Sequence (Sys_Sequence)

This class is a container that holds data items (uvm_sequence_items) which are sent to the driver via the sequencer. This class was extended from uvm_sequence and registered in the factory. uvm_sequence class is parameterized with uvm_sequence_item, so the Sys_Sequence is also parameterized with Sys_Sequence_Item. Finally, we implemented the body method which is used in the handshaking mechanism between the sequence and the driver.

5.2.5.1.8 System Test (Sys_Test)

This class was extended from uvm_test and register it in the factory. We declared environment and sequence handle. We wrote the new() function. Since the test is an uvm_component. The new() function has two arguments as string name and uvm_component parent. We implemented

the build_phase to create instances of environment and sequence classes. We implemented the run_phase to start sequences on required sequencers with raise/drop objection callbacks.

5.2.5.1.9 System Top (sys_top)

First, we set the timescale then import the uvm package. And then set the configuration parameters. The top is a static container that has an instantiation of DUT and interfaces. The interface instance connects with DUT signals in the top module. An interface is stored in the uvm_config_db using the set method and it can be retrieved down the hierarchy using the get method. UVM top is also used to trigger a test using run_test() call.

5.2.5.2 Register File Agent (RF_Agent)

As we mentioned in (section 3.5.2 Register File Interface) there are some registers inside the RF that can be read only, and others can be read and write. The main importance of the RF is to set some signals for the initialization process and read some signals for example, link_up which indicate that Link is ready for operation etc. This agent is used when we need it throughout the runtime.

5.2.5.2.1 RF Sequence item (RF_Sequence_Item)

As we mentioned above there are some inputs and some output signals for the RF interface, so the sequence item is defined to hold random inputs will be randomized inside the sequence before it is sent to the driver and some output signals not randomized.

5.2.5.2.2 RF Driver (RF_ Driver)

Everything here is as the system driver except the drive() task; it is responsible for converting the sequence item to pin level and drive it to the DUT at a specific time, see the code for more information.

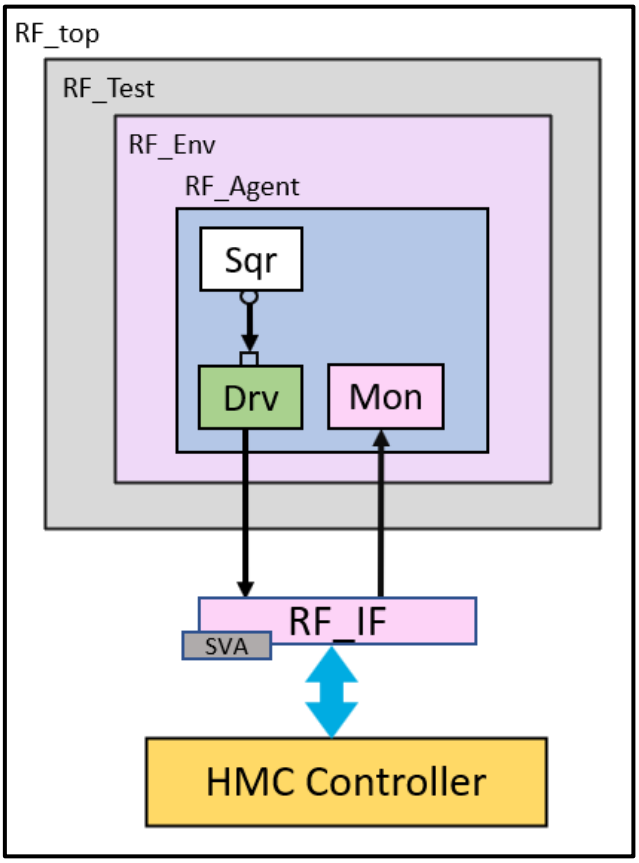


Figure 5. 6: RF Agent in UVM Env

5.2.5.2.3 RF Sequence (RF_Sequence)

We built a lot of sequences that can be used, in the following table we will mention a brief about the most important sequences.

Sequence	Description
RF_Disable_scrambler_Sequence	Write sequence. It is used to Disable Scrambler and Descrambler for testing purposes Disable the run length limiter.
RF_Set_P_RST_N_Sequence	Write sequence. It is used to set P_RST_N signal.
RF_Set_init_cont_Sequence	Write sequence. It is used to Allow descramblers to lock.
RF_Read_Status_Init_Sequence	Read sequence. It is used the read the status of the controller during the initialization process.

Table 5. 1: RF Sequences

All the remaining components and objects are the same as the system agent with some differences.

5.2.5.3 AXI Request Agent (AXI_Req_Agent)

The AXI-Request agent consists of several components responsible for applying multiple transactions from the testbench to the DUT, allowing for the testing of different features.

1.Sequencer: the sequencer in UVM serves as a critical component for coordinating and controlling the flow of transactions in a testbench. It enables efficient and structured transaction-based verification, helping to verify the functionality and behavior of the DUT while abstracting the complexity of the underlying transaction management.

2.The driver works closely with the sequencer to facilitate the transmission of transactions from the testbench to the DUT. It provides the necessary low-level translation and control to establish communication with the DUT's interface, ensuring proper protocol compliance and synchronization.

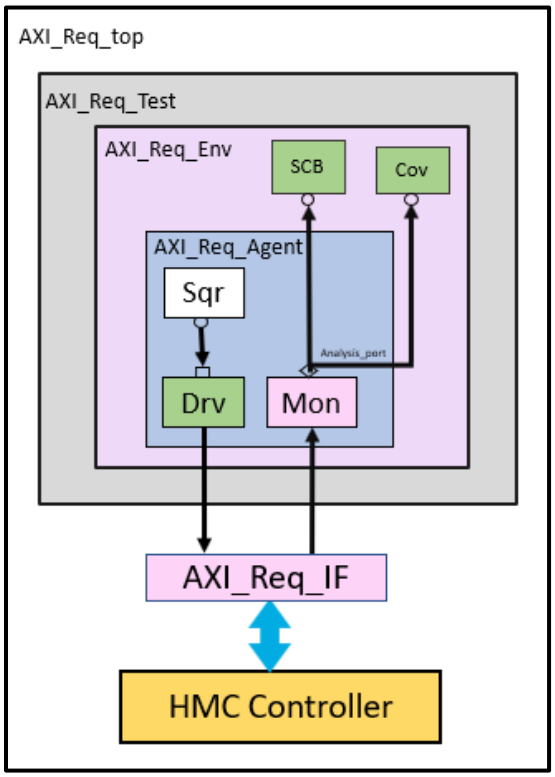


Figure 5. 7: AXI Request Agent in UVM Env

3.The monitor component plays a crucial role in capturing, analyzing, and verifying the behavior of the DUT. It provides visibility into the DUT's interfaces, enabling verification engineers to perform checks, coverage analysis, and debugging tasks to ensure the correctness and compliance of the design with the specified requirements.

5.2.5.3.1 AXI_Req_Sequence_item

In our project the data is the packet which contains the Header original data and header each packet represented as number of flits each flit consist of 128 bits. Header and Tails were explained.

So these packet are encapsulated in sequence item so it will be stimulus to the driver where the data derived to DUT to work with bins we pre-defined it

The Driver Responsible for Translating these data into Tuser and Tdata

The Content of the sequence Item :

- Commands Field which Has the fields Need for the request command like write Requests , Mode write Request , Read Requests and Read Write Requests
- The fields Of the Request Packet : The Fields of The Request Packets Needed To be transmitted.
- Constraints : Every sent Packet Should Be limited So it suitable for The Specs
- Built in Function : The Built in function like clone , Compare and Clone which needed for General Use and Uvm_object Is Responsible for it as Discussed below.

5.2.5.3.2 AXI_Req_Sequences

We have multiple AXI_Req_Sequences .

1)AXI_Req_Sequence: this is the base sequence .

2)Read16_Sequence: generates sequence with RD16 command and length of 1.

3)Read32_Sequence: generates sequence with RD32 command and length of 1.

4)Read48_Sequence: generates sequence with RD48 command and length of 1.

5)Read64_Sequence: generates sequence with RD64 command and length of 1.

6)Read80_Sequence : generates sequence with RD80 command and length of 1.

7)Read96_Sequence: generates sequence with RD96 command and length of 1.

8)Read112_Sequence: generates sequence with RD112 command and length of 1.

9)Read128_Sequence: generates sequence with RD128 command and length of 1.

10)Mode_Read_Sequence: generates sequence with MD_RD command and length of 1.

11)Write16_Sequence: generates sequence with WR16 command and length of 2.

12)Dual_8byte_add_Sequence: generates sequence with TWO_ADD8 command with a length of 2.

13)add_16byte_Sequence: generates sequence with ADD16 command with a length of 2

14)Posted_Write16_Sequence: generates sequence with P_WR16 command with a length of 2

15)Posted_Dual_8byte_add_Sequence: generates sequence with P_TWO_ADD8 command with a length of 2 16)Posted_add_16byte_Sequence: generates sequence with P_ADD16 command with a length of 2

17)Write32_Sequence: generates sequence with WR32 command with a length of 3

18)Posted_Write32_Sequence: generates sequence with P_ WR32 command with a length of 3

19)Write48_Sequence: generates sequence with WR48 command with a length of 4

20)Posted_Write48_Sequence: generates sequence with P_ WR48 command with a length of 4

21)Write64_Sequence: generates sequence with WR64 command with a length of 5

22)Posted_Write64_Sequence: generates sequence with P_ WR64 command with a length of 5

23)Write80_Sequence: generates sequence with WR80 command with a length of 6

24)Posted_Write80_Sequence: generates sequence with P_ WR80 command with a length of 6

25)Write96_Sequence: generates sequence with WR96 command with a length of 7

26)Posted_Write96_Sequence: generates sequence with P_ WR96 command with a length of 7.

27)Write112_Sequence: generates sequence with WR112command with a length of 8

28)Posted_Write112_Sequence :generates sequence with P_ WR112 command with a length of 8

29)Write128_Sequence: generates sequence with WR128 command with a length of 9

30)Posted_Write128_Sequence :generates sequence with P_ WR128 command with a length of 9

5.2.5.3.3 AXI_Req_Driver

This driver is responsible for converting transaction level such as fields of header and fields of tail to pin level like TDATA and TUSER and sending these signals to the HMC Controller.

- The AXI_Req_Driver class extends uvm_driver class, then registers the class with the UVM Factory by using `uvm_component_utils, ,then defines parameter FPW (flit per word) with a default value of two, then declares a virtual interface (axi_req_vif), then declares an instance of type AXI_Req_Sequence_item.

During the run phase, create a new instance of AXI_Req_Sequence_item, then call the get_next_item() method to retrieve the next item from the sequence, then call the drive_item() task, then call the item_done() method.

Drive_item() task:

Waits for positive edge of clock then check if reset is active low, set TDATA ,TUSER and TVALID to zero. If reset is high, wait for positive edge of clock ,then check if TREADY is high then call CREATE_PACKET_TDATA_TUSER() task then waits for positive edge of clock and finally sets TVALID to zero to indicate the completion of driving the packet, as shown in Figure 5.8.

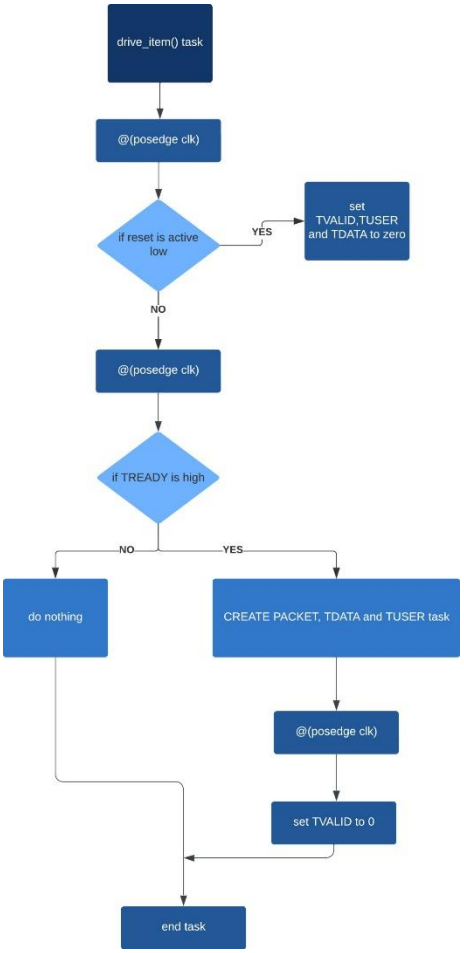


Figure 5. 8: AXI Request Driver (Drive Task)

CREATE_PACKET_TDATA_TUSER() task: This task generates a packet, which consists of flits. The number of flits in a packet is determined by the length of the packet.

1) creates the header of the packet and the tail of the packet. The tail of the request pkt must be all zeros.

2) There are four types of flits, as shown in the following figure: Flit size is 128 bits, header size is 64 bits, and Tail size is 64 bits.

3) Depending on LNG (LNG varies from 1 to 9),

When LNG equals one, indicating that the packet contains no data flits and the packet is a header and tail flit. and push back this flit to flits queue. TUSER fields{tail,hdr,valid} are set to one indicating valid data.

TAIL HEADER

-when LNG equals two ,indicating that the packet contains 2 flits. The first flit is{data0,header} and push back it to flits queue. . TUSER fields{tail,hdr,valid} are set to{0,1,1}. The second flit is[tail,data1} and push back it to flits queue. . TUSER fields{tail,hdr,valid} are set to{1,0,1}.

TAIL Data1	DATA0 HEADER

-default of case statement: the first flit is {data,header}.push back this flit to flits queue and TUSER fields{tail,hdr,valid} are set to{0,1,1}.then the loop iterates while i is less than or equal (axi_req_item.LNG-2)*2 . in each iteration flit={data[i],data[i+1]} and push back this flit to flits queue . set tuser fields to{0,0,1}.after the loop the final flit is{tail,data[(LNG-2)*2+1]}.push back this flit to flits queue. Set tuser fields to{1,0,1}.

-the while loop iterates while flits queue is not emplty and the number of cycles is less than max cycle

-FPW(flit per word) defines no of flits per cycle. If FPW value is 2,indicating that TDATA is two flits per cycle. If FPW value is 4,indicating that TDATA contains 4 flits per cycle.

AXI Request Data Transmission for WR80 Command

CYCLE	0	1	2
FPW	2		
LNG	6		
CMD	WR80		
FLIT1 TDATA[255:128]	DATA2 DATA1	DATA6 DATA5	TAIL DATA9
FLIT0 TDATA[127:0]	DATA0 HDR	DATA4 DATA3	DATA8 DATA7
Tail TUSER[5:4]	2'b00	2'b00	2'b10
Hdr TUSER[3:2]	2'b01	2'b00	2'b00
Valid TUSER[1:0]	2'b11	2'b11	2'b11
TUSER[5:0] {Tail, Hdr,Valid}	0x07	0x03	0x23

Figure 5. 9: AXI Request Data Transmission for WR80 CMD

The following table provides information about the transmission of an AXI request packet. It includes details such as the cycle number, FPW (flit per word), LNG (packet length), CMD (command), and the contents of each flit and TUSER field.

CYCLE	0	1	2	3
FPW	2			
LNG	2	1		
CMD	WR16	RD16	WR32	
FLIT1 TDATA[255:128]	TAIL DATA1	DATA0 HDR	TAIL DATA3	
FLIT0 TDATA[127:0]	DATA0 HDR	TAIL HDR	DATA2 DATA1	TAIL HDR
Tail TUSER[5:4]	2'b10	2'b01	2'b10	2'b01
Hdr TUSER[3:2]	2'b01	2'b11	2'b00	2'b01
Valid TUSER[1:0]	2'b11	2'b11	2'b11	2'b01
TUSER[5:0] {Tail, Hdr, Valid}	0x07	0x03	0x03	0x23

Figure 5. 10: AXI Request Packets Transmission

5.2.5.3.4 AXI_Req_Monitor

It is responanle for converting the pin level to transaction lwvel. It take TDATA and TUSER from the controler and accirding to it the response sequence item will be generated and it will broadcast the sequence item to the scoarboard and the coverage collecor.

5.2.5.4 HMC Memory Agent (HMC_Mem_Agent)

- **Introduction about Reactive Agent**

Verification components in a modern constrained-random setting can normally therefore be described as: Proactive Master: which initiates traffic to the DUT using sequence-based stimulus. Reactive Slave: which responds to traffic from the DUT using sequence-based stimulus. For a proactive master agent, the stimulus is explicitly initiated by the test sequences, resulting in a request being sent to the DUT from the proactive master agent, and the DUT generating the response in accordance with the protocol. For a reactive slave agent, the DUT is provoked into generating a request because of some remote stimulus or implicit behavior, the DUT generates the actual request whenever it wants, and the reactive slave agent generates a corresponding response in accordance with the protocol and verification requirements. This is our case, so we will explain the different architectures found to the reactive slave agent.

First, the reactive slave agent waits for a request from DUT and according to this request it will generate a response and drive it to the DUT.

- **Operations of HMC Reactive Slave Agent**

In our reactive slave agent, the monitor receives a request packet coming from the controller uplink, and then changes the pin level into transaction level (request packet sequence item), and then the monitor sends this request packet sequence item to the scoreboard to do its checks on the request packet sequence item.

The monitor also sends the request packet sequence item to the memory to process the required command of this request packet sequence item, and then the memory generates the suitable response based on the request sequence item command and puts this response packet in response packet sequence item to send it to the sequencer.

The sequencer sends request packet sequence item transaction to the driver to reform it again into pin level and send it back to the controller receiver through the controller downlink.

Reactive agent parts: (HMC_Mem_Slave_Agent)

- *Interface:* HMC_Mem_interface
- *Sequence Item:* HMC_Req_Sequence_item, HMC_Rsp_Sequence_item
- *Monitor:* HMC_Mem_Monitor.
- *Storage:* HMC_Mem_Storage
- *Sequencer:* HMC_Mem_Sequencer
- *Driver:* HMC_Mem_Driver
- *Agent:* HMC_Mem_Agent
- *Scoarboard:* HMC_Mem_Scoreboard
- *Environment:* HMC_Mem_Env
- *Test:* HMC_Mem_Test

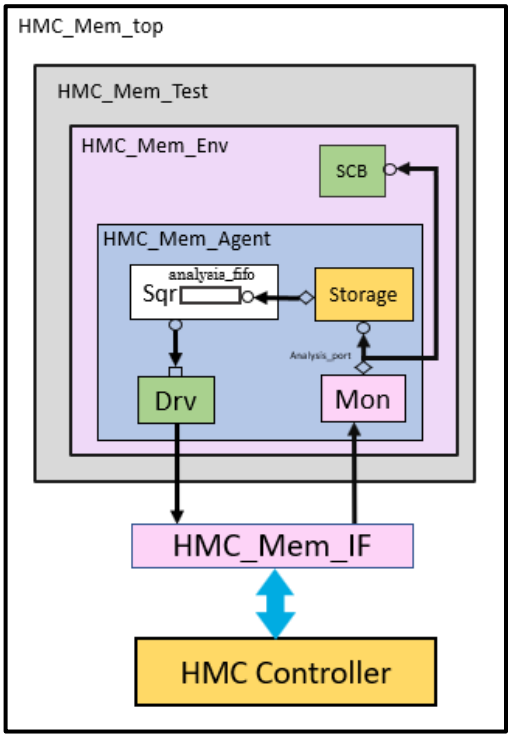


Figure 5. 11: HMC Mem Agent in UVM Env

5.2.5.4.1 HMC Memory Sequence item (Request and Response)

We have two sequence items: Request sequence item ”HMC_Req_Seq_item “ and Response packet sequence item “HMC_Rsp_Seq_item”.

As we mentioned above there are some inputs and some output signals for the Reactive slave agent interface and generally, the sequence item is defined to hold random inputs will be randomized inside the sequence before it is sent to the driver and some output signals not randomized.

The Request Sequence Item is defined to hold the transaction level for the incoming packets from the controller transmitter to the reactive agent monitor.

The Response Sequence Item is defined to hold the transaction level for the response packets generated from the memory of the reactive slave agent that went through the sequencer and finally driven to the controller receiver (in pin level at the end).

5.2.5.4.2 HMC Memory Monitor (HMC_Mem_Monitor)

As we are in reactive agent so the monitor has main functionality which it must do, and also it has different functionality as the mode of operation varies.

The main functionality for the monitor: the monitor receives, or sniffs request packets coming from the controller uplink and collects its FLITS whether in one cycle or in many cycles (based

on the request packet length and the rate in which the controller sends some FLITs “ FPW{FLITs Per Word} “).

After the HMC_Mem_Monitor collects all flits of the incoming request packet, it retransforms the packets from the pin level form into transaction level (request packet sequence item) form, and then the monitor sends this request packet sequence item to the scoreboard and the reactive slave agent memory.

The functionality of the HMC_Mem_Monitor in each mode of operation:

- **Monitor in Normal Mode:**
  - It does its main functionality, but when it catches a flow from the controller it will collect the packet from this flow, then it will check if this packet is correct or poisoned.
    - If it is correct, it will send it to the memory and the scoreboard.
    - Else it will go into HMC link retry mode.
- **Monitor in the initialization mode:**
  - The monitor is required to find out the sequence packets of the initialization mode and alarm the driver to send the required initialization packets in the right time and the correct sequence to complete the initialization operation.
- **Monitor in the Sleep mode:**
  - The monitor required to detect the sleep mode and alarm the reactive agent that we are going into sleep mode or not.
  - And also required to detect if the controller is getting out of the sleep mode, so it will alarm the reactive slave agent to leave sleep mode and transact to the initialization mode.
- **Monitor in the link retry mode:**
  - When the monitor detects an error in the incoming request packet, it will go through a sequence of steps to force the controller to retransmit the defected packet again.

The run_phase in the monitor is divided based on the current mode of operation.

1- At first in the initialization mode, it calls a function called “initialization_mode_operation”.
2- Then after finishing the initialization mode, it went into normal mode, so it calls a function called “normal_mode_operation”.
3- If the monitor detected an error, it would call a function called “link_retry_operation”.
4- Finally, if the controller went to sleep mode, the monitor will detect this, and it will call a function called “sleep_mode_operation”.

Steps of initialization operation inside initialization_mode_operation function:

1- Check on p_rst_n == 1.
2- Receive NULL1 packets.
3- Receive TS1 packets.
4- Within 1us TS1 packets should be sent from the driver.
5- Will receive NULL2 packets until the receiver starts to send TRET packets.
6- Start to receive TRET packets.
7- Finally we are in the active mode, so link_on = 1

The phases of normal_mode_operation function:

1- 1st phase: sniffing for upcoming data from the controller TX.
2- 2nd phase: storing the hall packet into a queue.
3- 3rd phase: dequeue the stored packet and send it to the memory.
4- 4th phase: to detect whether the received packet is correct, or it is not correct, and we need to go to link retry mode.

Steps of link retry mode inside link_retry_operation function inside monitor:

1- Detect error from "Cycle 1".
   A- Sequence-Number error.
   B- CRC error.
   C- LEN error
2- Start error flag.
3- in Cycle 2:
   A- the driver should send "Start_retry pulse."
   B- The monitor should receive the coming packet from the controller (if there is upcoming packet)
4- in Cycle 3:
   A- The monitor should not receive any new packets.
   B- The monitor should be waiting to "Clear Error packets" and count to 16 packets.
5- After receiving the 16 Clear Error packets, the monitor should be waiting to receive the retried packet.

Inside sleep_mode_operation function: The monitor sets LXRXPS and LXTXPS signals to 0

Link-Retry Mode in Reactive agent:

- **Some main definitions:**
  1- Some specifications for IRTRY packet:

- Seq_num =0
- CMD = 6'b000011
- RTC field =0

2- The FRP in the IRTRY packet works with new functionality, as:

- If (FRP[1] == 1) the IRTRY packet turns into:
  ClearError packet
  {Coming from the controller into the Reactive agent}
- If (FRP[0] == 1) the IRTRY packet turns into:
  StartRetry packet
  {Sent by the driver of the reactive agent to the controller}

3- IRTRY threshold

- The default is 16 packets {if the reactive agent driver sent 4 IRTRY packets to the controller}
- IRTRY receive threshold == 4 * number of IRTRY sent packets

Name	Start Bit	Size	Type	Reset Value	Description
Retry Status	0	1	RW	0x0	0x0: Retry is disabled 0x1: Retry is enabled
Retry Limit	1	3	RW	0x3	Controls the number of attempts before Link Error in ERRSTAT field indicates retry attempt limit has been met.
Retry Timeout Period	4	3	RW	0x5	0x0: 154ns 0x1: 205ns 0x2: 307ns 0x3: 384ns 0x4: 614ns 0x5: 820ns 0x6: 1229ns 0x7: 1637ns
Init Retry Packet Transmit Number	8	6	RW	0x08	0x2 – 0x3F: The number of IRTRY packets transmitted from Link Master during LinkRetry_Init is approximately 4 times the number specified in this field.
Init Retry Packet Receive Number	16	6	RW	0x10	The number of IRTRY packets that the link slave will detect before it clears Error Abort Mode.

Main checks which can launch Link-Retry mode

When the reactive agent receives a wrong packet which may have one of those errors:

1- Sequence number error.
Upon receipt of the packet, the link slave will verify that the sequence number has a +1- value compared to the last successful packet received.

2- CRC error.
Upon receipt of the packet, the link slave will remove the incoming packet CRC and store it somewhere, then it will recalculate the CRC for the hall packet again, finally it will compare the old CRC coming with the packet with the new CRC which it recalculated it.

- If the New CRC == the old CRC, the packet is correct.
- If the New CRC == one's complement of the old CRC, that means, this received packet is poisoned (START Link Retry Mode).
- If the New CRC != the old CRC that means, this received packet is a wrong packet (START Link Retry Mode).

3- LNG error.
Upon receipt of the packet, the link slave will verify that the LNG (the length of the packet) is the right length for the packet command.

Link-Retry mode operation steps:

a- We extract FRP and then we make the packet RRP == FRP
b- Error abort mode signal (inside the HMC).
c- Inserts a stream of IRTRY packets (with StartRetry flag set {FRP[0]=1})
d- Imbeds the last FRP into the RRP of each IRTRY packet, the value of RRP is the FRP of the last good packet received at the green link(Reactive agent monitor)
e- Sends StartRetry pulse (Driver) {the number of IRTRY packets should be received from the controller will be 4 times the number of IRTRY packets sent from the reactive agent driver into the controller}
So, we need to send like 4 IRTRY packets or 8 IRTRY packets.
f- the monitor will receive the rest of this packet, then it will receive IRYTRY packets from the controller, with ClearError Flag (FRP[1] =1)
g- when the monitor finds out IRTRY packets with ClearError, it will initiate a counter to count,
h- when the counter reaches the IRTRY threshold, it drops the error abort mode.
i- Then the controller will resend the packet again to the reactive agent.
j- The monitor will check again the CRC, sequence number, the LEN of the retried packet.
k- The retried packet should have sequence number = the last successful transmitted packet sequence number +1

Monitor operation in this part:

1- Detect error from "Cycle 1"

A- Sequence-Number error.

B- CRC error.

C- LEN error

2- Start error flag.

3- in Cycle 2:

A- the driver should send "StartRetry pulse."

B- The monitor should receive the coming packet from the controller (if there is upcoming packet)

4- in Cycle 3:

A- The monitor should not receive any new packets.

B- The monitor should be waiting to "ClearError packets" and count to 16 packets.

4- After receiving the 16 ClearError packets, the monitor should be waiting to receive the retried packet.

5.2.5.4.3 HMC Memory Storage (HMC_Mem_Storage)

The main functionality of the storage is to store the write request packet according to the address field that exists in the header of this packet. If the request packet is a read packet, the address of this packet will be searched as it may not be stored at any time before. It is also responsible for returning a response packet to match the request packet and update some of its fields. In the following table we will explain the storage functionality.

Request Packet		Response Packet
Packet with Invalid length LNG >= 10	-	A read response packet is generated. ERRSTAT = 7' b0110001
Write packet	It will be stored in the storage; its address will be the packet address that exists in the packet header.	A write response packet is generated. Some of its fields remain as the request packet and other will be updated.
Posted write packet	It will be stored in the storage, as the write packet.	Not waiting any response.
Read packet	It will be read from the storage.	A read response packet is generated. Some of its fields remain as the request packet and other will be updated.
Packet with Invalid command	-	A write response packet is generated. ERRSTAT = 7'b0110000
Link Retry packet	-	A response packet is generated. CMD = IRTRY

Table 5. 2: Storage Functionality

5.2.5.4.4 HMC Memory Driver (HMC_Mem_Driver)

It will perform the initialization as discussed in section 4.2.1, then it will drive the response packet that will be generated in the storage to phy_data_rx_phy2link pin. In the following we will discuss three cases of the packet where the FPW = 2, which means the phy_data_rx_phy2link signal width is 256 bits.

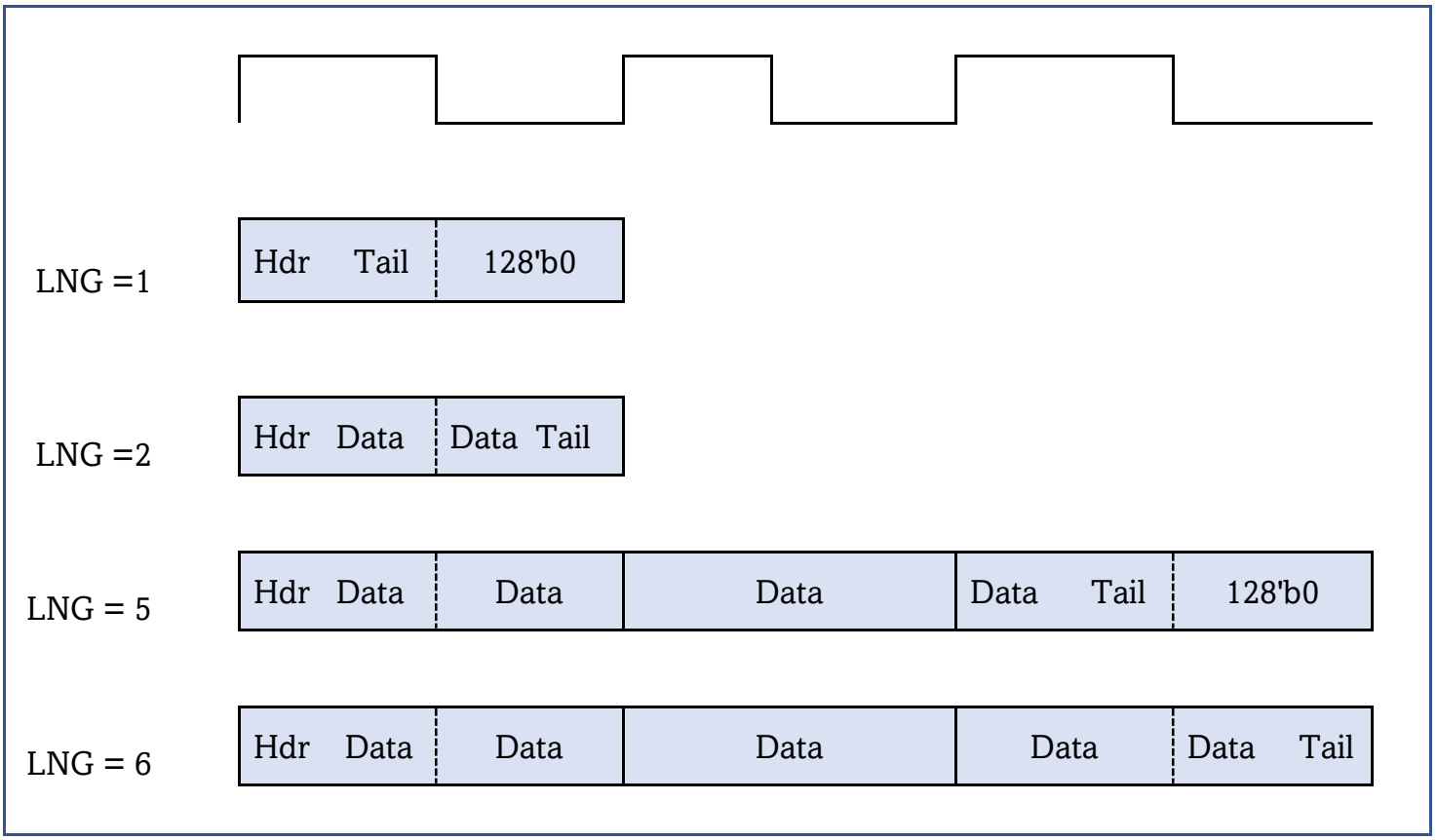


Figure5. 1: HMC Mem Driver

LNG =1 : The Packet length is 128 bits but the phy_data_rx_phy2link pin width is 256 bits, so the driver will fill the remaining bits with zeros and drive it to the pin, it will take one clock cycle.

LNG =2 : The Packet length is 256 bits and the phy_data_rx_phy2link pin width is 256 bits, so the driver will drive it to this pin directly and it will take one clock cycle.

LNG =5 : The Packet length is 640 bits but the phy_data_rx_phy2link pin width is 256 bits, so the driver will concatenate each 2 flits with each other and fill the remaining bits in the last one with zeros and finally drive them to the pin, it will take three clock cycle.

LNG =6 : The Packet length is 768 bits but the phy_data_rx_phy2link pin width is 256 bits, so the driver will concatenate each 2 flits with each and drive them to the pin, it will take three clock cycle.

5.2.5.4.5 HMC Memory Agent (HMC_Mem_Agent)

The HMC_Mem_Agent just connects the remaining components ports together. It connects:

1- Memory with storage.
2- Storage with sequencer.
3- Sequencer with Driver.

5.2.5.5 AXI Response Agent (AXI_Rsp_Agent)

This agent is used When the controller needs to send a response packet to the user application.

5.2.5.5.1 AXI Response Sequence item (AXI_Rsp_Sequence_Item)

The contents of a single packet, including its header, data, and tail, are stored as sequence items in this agent. The header and tail information are presented as fields. Additionally, the sequence item has a TREADY signal, which is involved in the handshake process between the controller and the user application.

To align with the specification requirements, we impose certain constraints on the randomization of certain fields. These constraints pertain to the length and commands, as each command has a predetermined number of FLITS. Additionally, we ensure that the randomized values for both the LNG and DLN fields are identical. Furthermore, we always set the reserved parts to zeros.

Command Description	Symbol	Packet Length in FLITs
READ response	RD_RS	1 + data FLITs
WRITE response	WR_RS	1
ERROR response	ERROR	1

Table 5. 3: Response Commands

5.2.5.5.2 AXI Response Driver (AXI_Rsp_Driver)

In this agent, the only signal that the driver is responsible for driving to the controller is the TREADY signal. During the run phase, the driver waits for a positive edge of the clock and checks whether the reset is active or not. If the reset is active, the driver also resets the TREADY signal. If the reset is not active, the driver drives the TREADY signal with the randomized values of the sequence.

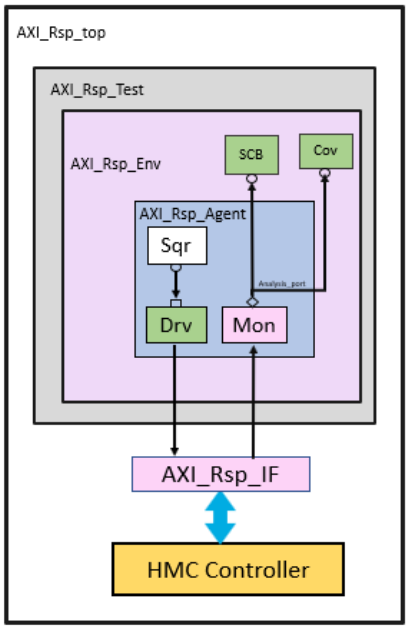


Figure 5. 12: AXI Response Agent in UVM Env

5.2.5.5.3 AXI Response Monitor (AXI_Rsp_Monitor)

The objective of the class is to observe TDATA and TUSER signals from the DUT and converting them to transaction level signals and forward them to the Monitor subscribers' classes.

Firstly, we use the uvm_component_param_utils macro to register the class in the factory. By using the factory, components can be created and configured, allowing for a highly flexible and modular verification environment.

Most of the Monitor's work occurs during the Run phase. Our concern here is to convert from the Pin Level used in the interface to the Transaction Level that other components can understand. We first start with instantiating object (item) from AXI_Rsp_Sequence_Item class using UVM factory.

TDATA and TUSER are Pin Level signals, whereas the item object represents the Transaction Level. Therefore, we must extract the required information from the TDATA and TUSER signals and store it in the item object which will be further sent to subscribers.

TDATA and TUSER signals are considered valid in the interface only when both TVALID and TREADY signals are high. This indicates that TDATA and TUSER signals are valid in the DUT and the user application is ready to sample them. At the positive edge of the clock, we confirm whether both signals are high. If they are, then we initiate the process of converting from Pin Level to Transaction Level. So how do we convert them?

In achieving this we used Pin_To_Trans, To_Header and To_Tail tasks, so we are going to explain the main function of each of them.

To_Header task:

We will begin by discussing To_Header task. This task takes two arguments: a 64-bit header variable and an object of the AXI_Rsp_Sequence_Item class named item. As we know, the

Transaction Level comprises a FLIT, which can contain header, tail, or data. Both the header and the tail comprise several fields. In this task, we only focus on the header data Recall that each header consists of 64 bits, but only a few of them are utilized in the fields: CMD, LNG, DLN, and TAG. Therefore, the primary function of this task is to take all 64 bits of the header data that we extracted from the Pin Level signals (stored in the header variable) and assign the necessary bits to the fields of the item object header.

It is evident that the assignment process for all fields is simple, except for the CMD field. The assigning of bits to the CMD field is different because it is an enumerated data type. Therefore, before assigning the bits to the CMD field, we must typecast them to the same data type.

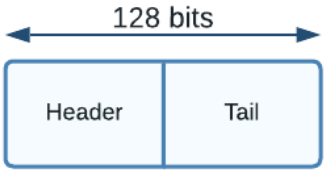
To_Tail task:

This task is very similar to the To_Header task, as they share the same concept. The task takes all 64 bits of the tail data extracted from the Pin Level signals (stored in the tail variable) and assigns the necessary bits to the tail fields of the item object.

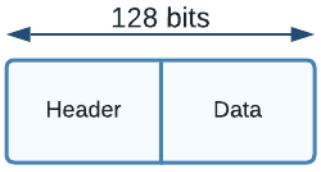
Pin_To_Trans task:

It is necessary to address all the possible combinations that may arise. A FLIT is composed of 128 bits, and it can be either a NULL FLIT or a Data FLIT. A NULL FLIT comprises 64 bits of Header and 64 bits of Tail. On the other hand, a Data FLIT can take one of the following three forms:

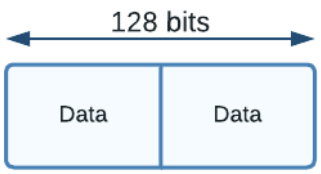
1- It can consist of 64 bits of header and 64 bits of data.
2- All the 128 bits can be data.
3- The first 64 bits can be data and the other 64 bits can be tail.



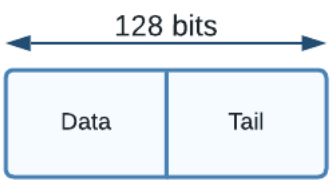
NULL FLIT



Data FLIT



Data FLIT



Data FLIT

Then we need to know about each 64 bits if they are header, data, or tail bits, how we can do that? It is the turn of the three flags in the interface Hdr, Tail and Valid flag. In each clock cycle the TDATA signal have FPW*128 bits, may not all of them can be FLITs as in that case the number of FLITS available in each clock can be less than or equal to FPW, so it is also the job of the three flags to tell us if there is a FLIT here or not, let us explain their job.

To determine whether each 64-bit portion belongs to the header, data, or tail, we use the three flags in the interface: Hdr, Tail, and Valid which are extracted from TUSER signal. Each clock cycle, the TDATA signal contains FPW*128 bits. Not all these bits can be a valid FLIT, which can result in the number of FLITS available in each clock being equal to or less than FPW. Therefore, the three flags are responsible for indicating whether a FLIT is present or not.

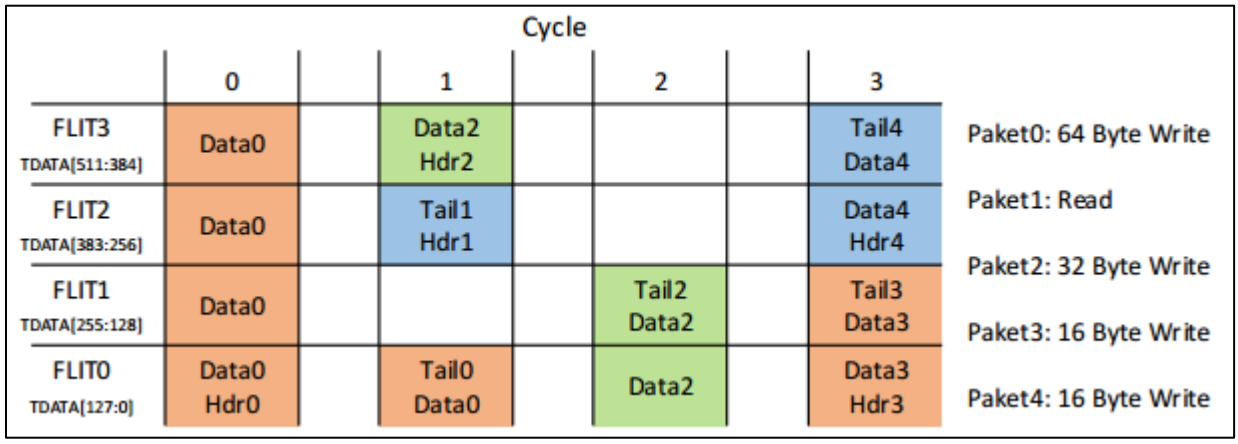


Figure 5. 13: Example on TDATA bus for FPW=4

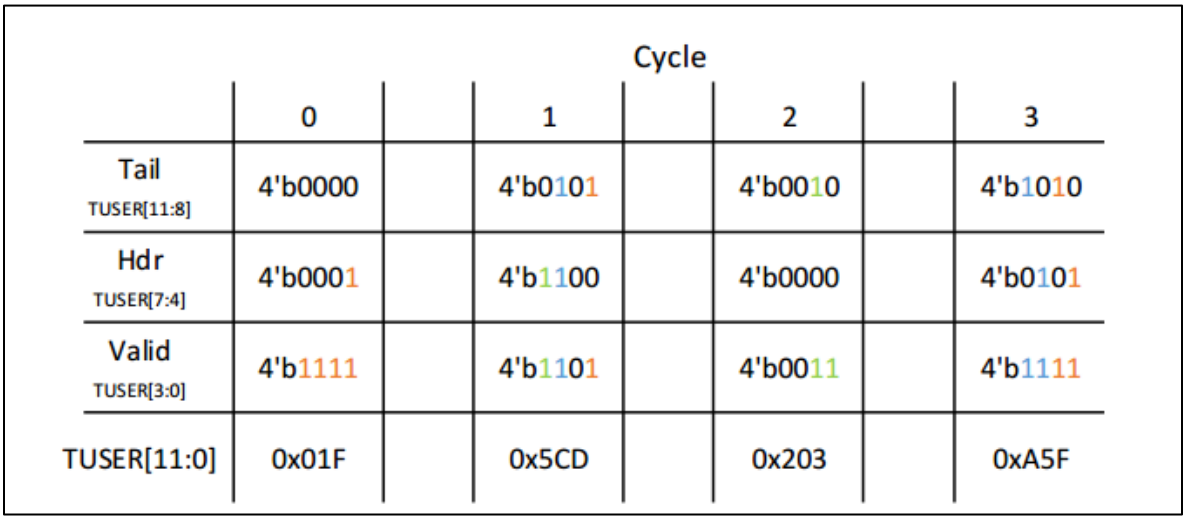


Figure 5. 14: TUSER for the pervious Example

Every FLIT on the TDATA bus corresponds to one bit in the valid, hdr, and tail fields on TUSER. FLIT0 at TDATA[127:0] is defined by valid[0], hdr[0],and tail[0].

Example: TDATA holds a header on FLIT position 0 (TDATA[127:0]). Set hdr[0] to high. Since a header is a valid FLIT, set valid[0] high. This scheme applies to all FLITs on the TDATA bus. Figure illustrates how to set the TUSER signal according to the content of the TDATA bus in Figure.

Let's return to Pin_To_Trans task, first we store the TDATA into array of 64 bits width.

Then we create a loop which iterates according to the FPW, representing the maximum number of FLITs that can be transmitted in a single cycle. The first check performed in the loop is to determine whether the FLIT is valid by checking whether the Valid flag is high.

After determining that the FLIT is valid, the next step is to identify whether it contains a header, tail, or data. If the Hdr flag is high, we know that the first 64 bits of the FLIT are the header data of a new packet. We then extract those bits from the array and call the To_Header task to fill in the header fields. However, we still need to determine whether the remaining 64 bits are data or tail bits. To achieve this, we examine the LNG field in the header, which informs us of the number of FLITs in the packet. If LNG equals 1, then the remaining 64 bits represent tail data, indicating a NULL FLIT. If LNG is greater than 1, then the bits represent data, and we assign them to the first element in the data array of the item object.

The next step is to check the Tail flag. If it is high, then we have received the final FLIT of a complete packet. We then call the To_Tail task with the last 64 bits of the FLIT. Before we assign the first 64 bits to the last element of the data array, we check whether it is a NULL FLIT. Since we have determined that we have a complete packet, we can use the write function to write data into the analysis port, which connects the Monitor and subscribers. However, if neither the Hdr nor the Tail flag is high, we know that all the FLITs are data bits.

5.2.6 Scoreboards

5.2.6.1 HMC Scoreboard (HMC_Scoreboard)

The main functionality of this scoreboard is to compare the Request Packets on both sides of the controller AXI Request with the HMC Request and compare the Response Packets at both sides of the controller AXI Response with HMC Response. Figure 5. 3 shows how this scoreboard is connected.

Request Packet Comparing Section:

Cube ID	Address	Tag	LNG	DLN	CMD	Data size

Response Packet Comparing Section:

Tag	LNG	DLN	CMD	Data size

5.2.6.2 AXI Request Scoreboard (AXI_Req_Scoreboard)

This scoreboard is responsible for verifying the correctness of various fields in the AXI request packet. it contains multiple tasks like

Check_LNG task: It verifies the correctness of the "CMD" and "LNG" fields in the header. It checks if the "CMD" field matches any of the specified commands and then validates if the corresponding "LNG" field has the expected value. If the conditions are met, an information message is printed using uvm_info(). Otherwise, an error message is printed using uvm_error().

Check_Tail task: It verifies specific fields, including "CRC", "RTC", "SLID", "SEQ", "FRP", and "RRP", in the tail. It checks if all these fields are zero and, if so, prints an information message using uvm_info(). Otherwise, it prints an error message using uvm_error().

Check_Header task: It verifies the equality of "LNG" and "DLN" fields, the values of "CUB" and "SLID" fields, and the size of the "data" field based on the "LNG" value.

5.2.6.3 HMC Response Scoreboard (HMC_Mem_Scoreboard)

The HMC_Mem_Scoreboard receives request packet sequence item from the monitor through analysis port, and the HMC_Mem_Scoreboard receives response packet sequence item from the memory.

It simply makes some checks on each sequence item and finally, it checks whether the response packet is the right response packet for the incoming request packet.

Scoreboard checks:

1- Request packet checks:
   1) Check if the command exists or not.
   2) Check if it is posted write request command or not.
   3) Check LNG with the CMD.
   4) Check the CRC of the request packets.

2- Common Checks:
   1) check if the request and response packets have the same (TAG, SEQ_NUMBER)
   2) Check if the response packet CMD is the right response CMD for the request packet CMD.
   3) Check if the request and response packets have the same (FRP, RRP).

3- Response packet checks:
   1) Check if the command exists or not.
   2) Check the CRC of the Response packets.

5.2.6.4 AXI Response Scoreboard (AXI_Rsp_Scoreboard)

The primary purpose of this scoreboard is to verify the compatibility of the header and tail fields with the expected values. Specifically, for the tail fields, the scoreboard checks that all fields contain zeros. For the header fields, the scoreboard checks whether the length of the packet is consistent with the response command received and confirms whether the values of both the LNG and DLN fields are identical or not.

5.2.7 Coverage Collectors

Subscribers are listeners of an analysis port. They subscribe to a broadcaster and receive objects whenever an item is broadcasted via the connected analysis port.
We have two subscribers : AXI_Req_Subscriber and AXI_Rsp_Subscriber.

5.2.7.1 AXI Request Coverage (AXI_Req_Subscriber)

In AXI_Req_Subscriber ,there are three cover groups:
The first cover group is axi_req_pkt_cov. :This cover group is responsible for ensuring coverage of all commands. there are 5 cover points:
1-WRITE_CMD_CP captures different write commands such as WR16, WR32, WR48, WR64, WR80, WR96, WR112, and WR128.
2-ATOMIC_CMD_CP captures atomic commands such as TWO_ADD8 and ADD16.
3-POSTED_WRITE_CMD_CP captures different posted write commands such as P_WR16, P_WR32, P_WR48, P_WR64, P_WR80, P_WR96, P_WR112, and P_WR128.
4-POSTED_ATOMIC_CMD_CP captures posted atomic commands such as P_TWO_ADD8 and P_ADD16.
5-READ_CMD_CP captures different read commands such as RD16, RD32, RD48, RD64, RD80, RD96, RD112, RD128, and MD_RD (mode read).
The second cover group is CMD_transitions: This cover group is responsible for covering transitions between different commands, there are 4 cover points:
1- write_read_write cover point captures the transitions where a write command (e.g., WR16, WR32) is followed by a read command (e.g., RD16, RD32), and then followed by another write command of the same type. It defines bins such as wrw16 for WR16 => RD16 => WR16 transition, wrw32 for WR32 => RD32 => WR32 transition, and so on for different command types.

2-read_write_read cover point captures the transitions where a read command (e.g., RD16, RD32) is followed by a write command (e.g., WR16, WR32), and then followed by another read command of the same type. It defines bins such as rwr16 for RD16 => WR16 => RD16 transition, rwr32 for RD32 => WR32 => RD32 transition, and so on for different command types.
3-posted_write_read_write cover point captures the transitions where a posted write command (e.g., P_WR16, P_WR32) is followed by a read command (e.g., RD16, RD32), and then followed by a write command (e.g., WR16, WR32) of the same type. It defines bins such as pwrw16 for P_WR16 => RD16 => WR16 transition, pwrw32 for P_WR32 => RD32 => WR32 transition, and so on for different command types.
4-(postedwrite_read_postedwrite) cover point captures the transitions where a posted write command (e.g., P_WR16, P_WR32) is followed by a read command (e.g., RD16, RD32), and then followed by another posted write command of the same type. It defines bins such as pwrw16 for P_WR16 => RD16 => P_WR16 transition, pwrw32 for P_WR32 => RD32 => P_WR32 transition, and so on for different command types.
The third cover group is CMD_Consecutive: This cover group is responsible or covering consecutive commands .there are 4 cover points:
1-write_write_write cover point captures consecutive write commands of the same type. It defines bins such as www16 for WR16 => WR16 => WR16 sequence, www32 for WR32 => WR32 => WR32 sequence, and so on for different command types.
2-write_write_write cover point captures consecutive write commands of the same type. It defines bins such as www16 for WR16 => WR16 => WR16 sequence, www32 for WR32 => WR32 => WR32 sequence, and so on for different command types.
3-postedwrite_postedwrite_postedwrite coverpoint captures consecutive posted write commands of the same type. It defines bins such as pwrw16for P_WR16 => P_WR16 => P_WR16 sequence, pwrw32 for P_WR32 => P_WR32 => P_WR32 sequence, and so on for different command types.

5.2.7.2 AXI Response Coverage (AXI_Rsp_Subscriber)

There is one cover group(axi_rsp_pkt_Cov)was created.

Several cover points are defined in this cover group:

1) Flow_CMD_CP focuses on the CMD field of the AXI response packets, and it captures bins like (Null, Retry_pointer_return,Token_return, and Init_retry)

2) Response_CMD_CP focuses on the CMD field and captures bins like READ_response, WRITE_response, ERROR_response, MODE_READ_response, and MODE_WRITE_response.

3) PACKET_LENGTH cover point covers the LNG field of the AXI response packets and defines bins using the range [4'b0001:4'b1001] to represent different packet lengths in FLITs.

4) Duplicate_LENGTH cover point focuses on the DLN field and captures bins like the PACKET_LENGTH cover point.

Chapter 6: Simulation Results and Coverage

In this chapter, we are going to discuss the results obtained from the simulation of the environments and the bugs detected by tracing the reason for the bug and possible solutions if one was found. We will start simulating the System Agent, Register File Agent, and the HMC Memory Agent (Reactive Agent) and then simulating the system overall.

6.1 System Agent, RF Agent, and the HMC-Mem Agent Simulation

6.2 AXI Request Simulation

6.3 AXI Response Simulation

6.4 HMC-Mem Agent Simulation

6.5 Bugs

Chapter 7: Conclusion and Future Work

7.1 Github link

https://github.com/mohamedibrahem399/HMC_Controller_Verification

7.2 Graduation project phases

1) Phase one: (Learning main concepts).
2) Phase two: (Reading Specification and documentation).
3) Phase three: (Making Verification plan and architecture).
4) Phase four: (Coding phase).
5) Phase five: (Compilation and simulation for each part alone).
6) Phase six: (Combining all agents and start running our tests).

7.3 Conclusion

By the end of this project, we now fully understand the digital design verification flow and
developed our design verification skills that lead us to achieve our goals and successfully build two verification environments designed using SystemVerilog and UVM methodology
to verify any HMCC (Hybrid Memory Cube Controller).

7.3.1 What we learnt during this graduation project:

1) OOP using C++.
2) System Verilog
3) Making full verification for ALU with System Verilog verification concepts.(Project)
4) UVM using System Verilog.
5) How to make verification plan.
6) How to make UVM architecture.
7) How to compile and simulate UVM environment and how to add tests to this environment.
8) UVM assertions.

7.4 FUTURE WORK

1- Parameterized UVM classes.

- We made our classes parameterized, but in the integration we had to set those parameters to specific values and we tried the project with those sat parameters.
- So you can try to run the environment with different parameters.

2- Usage of different clocks.

- In OPEN_HMCC, there is a feature in which the OPEN_HMCC can run in different clocks, the first clock for the AXI part, and the second clock for the remaining OPEN_HMCC components.
- You can test this feature in different modes.
    - In initialization mode.
    - In normal mode.
    - In Tx mode
    - In link retry mode.
- And especially in a specific mode (Link retraining mode).
    - This mode detects unacceptable rate of link error.
    - This process is monitored by the link_retries counter.
    - In this mode, the HMCC should enter sleep mode and exited to retrain the link.
    - All the steps described in section 4.3 in the OpenHMC_specification.

3- Making internal checks for each block of the controller internal blocks.

- We made our tests based on the pin level and the interfaces of the OPEN_HMCC design.
- You can make specific tests for each block inside the OPEN_HMCC.

4- ASYNC input FIFO.

- Inside the OPEN_HMCC there are
    - ASYNC/SYNC input FIFO.
    - ASYNC/SYNC output FIFO.
- In our testing we used the SYNC mode for those FIFOS, so you can test the ASYNC mode for those FIFOS.

5- Making a chain of memory cube and connect them with our controller.

- We made a reactive slave agent which plays the role of the HMC memory cube and react with the OPEN_HMCC (the controller), but we made the reactive agent as it has just one memory cube.
- You can add more memory cubes inside the reactive slave agent and try to connect them together and test the system.
- For more information check the HMC_Specification 1_0

References and Resources

- Specifications and documents

1. openHMC_documentation_1.5
   https://github.com/unihd-cag/openhmc/blob/master/doc/openHMC_documentation_1_5.pdf
2. HMC_Specification 1_0
   https://www.nuvation.com/sites/default/files/Nuvation-Engineering-Images/Articles/FPGAs-and-HMC/HMC-30G-VSR_HMCC_Specification.pdf

- Si-Vision tutorial videos (On Classroom)

3. OOP using C++
4. System Verilog
5. UVM Methodology using system Verilog.

- References

6. Writing Testbenches Functional Verification of HDL Models
   https://link.springer.com/book/10.1007/978-1-4615-0302-6
7. UVM Primer.
   https://drive.google.com/file/d/1YyLMw-573_a8ATFFsytPrKAI_J6CZYn6/view?usp=sharing
8. Coverage-cookbook:
   https://picture.iczhiku.com/resource/eetop/wYIdrgQtajwzzNmC.pdf
9. uvm_users_guide_1.2
   https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.1.pdf

- Scientific papers

10. CummingsDVCon2016_Vsequencers
    http://www.sunburst-design.com/papers/CummingsDVCon2016_Vsequencers.pdf

11. CummingsDVCon2020_UVM_ReactiveStimulus
http://www.sunburst-design.com/papers/CummingsDVCon2020_UVM_ReactiveStimulus.pdf
12. CummingsSNUG2009SJ_SVA_Bind
http://www.sunburst-design.com/papers/CummingsSNUG2009SJ_SVA_Bind.pdf
13. CummingsSNUG2013SV_UVM_Scoreboards
http://sunburst-design.com/papers/CummingsSNUG2013SV_UVM_Scoreboards.pdf
14. MASTERING REACTIVE SLAVES IN UVM by (Mark Litterick , Jeff Montesano , Taruna Reddy)
https://www.verilab.com/files/mastering_reactive_slaves.pdf
15. UVM Reactive agents verify with a handshake
https://www.edn.com/uvm-reactive-agents-verify-with-a-handshake/

- Websites

1. ChipVerify
https://www.chipverify.com/
2. Verificationacademy
https://verificationacademy.com/
3. testbench
http://testbench.in/CO_09_TRANSITION_BINS.html
4. VSLI Verify
https://vlsiverify.com/
5. Accellera
https://forums.accellera.org/
6. Verlab
http://www.verilab.com/resources/papers-and-presentations/#slaves2
7. UVM kit
https://uvmkit.com/reactive_agents.html
8. Sunburst
http://www.sunburst-design.com/papers/
9. UVM the University of Vermont
https://www.uvm.edu/directory