Implementation of Hybrid Memory Cube Interface

A Project Report

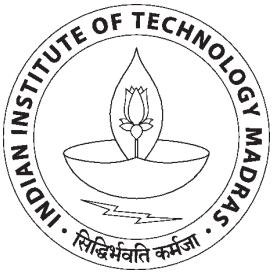
submitted by

GADDAM BHARATH KUMAR

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

MASTER OF TECHNOLOGY

under the guidance of
Dr. V. Kamakoti



DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.

MAY 2015

THESIS CERTIFICATE

This is to certify that the thesis entitled **Implementation of Hybrid Memory Cube Interface**, submitted by **Gaddam Bharath Kumar**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bonafide record of the research work carried out by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

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

Place: Chennai

Date:

ACKNOWLEDGEMENTS

I Would like to express my deepest gratitude to my guide, ***Dr. V. Kamakoti*** for his valuable guidance, encouragement and advice. His immense motivation helped me in making firm commitment towards my project work.

My special thanks to ***Mr. G.S. Madhusudan*** for his encouragement and motivation through out the project. His valuable suggestions and constructive feedback were very helpful in moving ahead in my project work.

I would like to thank my faculty advisor ***Dr. Nagendra Krishnapura*** who patiently listened, evaluated, and guided us through out our course.

My special thanks to my fellow labmates Neel, Sukrat, Lakshmeesha, Anand, Gopinathan for their help and support.

ABSTRACT

KEYWORDS: Hybrid Memory Cube, Requester, Link Retry, FLIT and CRC

Hybrid Memory Cube (HMC) is new DRAM technology. HMC scores better in terms of power, area and performance compared to current DDR SDRAMs. So it is said, HMC is going to be the future of DRAMs. As the data transmission techniques followed by HMC are completely different from DDR SDRAMs, we need a new interface block to connect the processors with HMC. My project work involves building a circuit which interfaces the processor with HMC. This involves study of data transmission mechanism used by HMC and implementation of interface block using the same.

The interface block being on the processor side can be called Requester as it sends the requests to memory. As per the HMC data transmission protocol, Requester block consists of three layers namely Transaction, Link and Physical layers. The Requester block can also be divided into two blocks namely Sending and Receiving block. In addition to them, Requester also consists of a Retry block to take care of transmission failures. Entire Requester block is implemented. A Test Bench is also written to verify the working of the Requester block. The test bench involved creating read and requests to Memory and printing out the Responses from Memory. The code for the entire project is written in a HDL namely Bluespec System Verilog(BSV).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

HMC	Hybrid Memory Cube
TSV	Through Silicon Via
DRAM	Dynamic Random Access Memory
BSV	Bluespec System Verilog
HDL	Hardware Description Language
FLIT	Flow Unit
LFSR	Linear Feedback Shift Register
CRC	Cyclic Redundancy Check
FRP	Forward Retry Pointer
RRP	Return Retry Pointer
RTC	Return Token Counts
PRET	Retry pointer return
TRET	Token Return
IRTRY	Init Retry

CHAPTER 1

INTRODUCTION

1.1 Overview

A group of people in Reconfigurable and Intelligent Systems Engineering (RISE) Lab in the Computer Science Dept. of IIT Madras are actively involved in building processors for various applications. They want their processors to have support for Hybrid Memory Cube(HMC), as HMC is going to be upcoming DRAM technology. So I am given a task to build an interface block that connects processor with HMC. The interface block needs to have both sending and receiving sections. Sending section receives read/write requests from processor and converts them to packets and sends them to HMC. The receiving section will take care of extracting packets received from HMC, and sending the extracted read/write responses to the processor. As per the HMC data transmission protocol, Requester block can also be divided into 3 layers namely transaction, link and physical layers. The packets pass through all the 3 layers in both directions.

1.2 Objective

Main focus of the project is to implement both sending and receiving sections of the interface block. It should support the retry mechanism to take care of transmission failures. The block needs to be designed by writing code in HDL namely Bluespec System Verilog. It needs to be synthesized and tested on FPGA which has HMC installed on it.

1.3 Organization of the thesis

Chapter 2 deals with architecture and configuration of Hybrid Memory Cube. It discusses its logic base configuration which consists of multiple serial I/O links. It also discusses bandwidth details of HMC.

Chapter 3 deals with data transmission mechanism used by HMC. It discusses the 3 layers of the link in detail. It discusses the constituents of Request and Response packets. It discusses error detection technique used by the protocol. It also describes the link retry feature that is used to to take care of transmission failures.

Chapter 4 gives us implementation perspective of the project. It explains in detail how each block of the interface is implemented in Bluespec System Verilog(BSV). It also describes how the Test Bench was written to verify the working of the interface module.

Chapter 5 contains a conclusion and description on the future work.

CHAPTER 2

HYBRID MEMORY CUBE

HMC is a new DRAM architecture that is being worked out by a consortium of many technology companies. The companies include Micron, Hynix, Samsung, Altera, Xilinx, IBM, ARM and Open Silicon. HMC promises higher access speeds when compared to the current DDR SDRAMs. It is said that HMC is going to replace the current DDR SDRAMs in near future.

2.1 Architecture

Architecture of HMC differs from conventional DRAMs. In HMC, several DRAM die are stacked one above the other to make a 3D structure, a cube. This structure will have a logic die at the base. As it contains both memory and logic die, it is called Hybrid Memory Cube. Generally in a single package of HMC, there will be either four or eight DRAM die and one logic die. All of them are stacked one above the other using through-Silicon Via(TSV) technology. TSV technology makes the connections between the die possible.

Memory is organized vertically in each cube of HMC. Memory layers are divided into partitions. Memory Partitions from various layers in a single stack combined with a corresponding controller in logic layer form a vault. Each vault is functionally and operationally independent.Each vault has a memory controller (called a vault controller) in the logic base 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

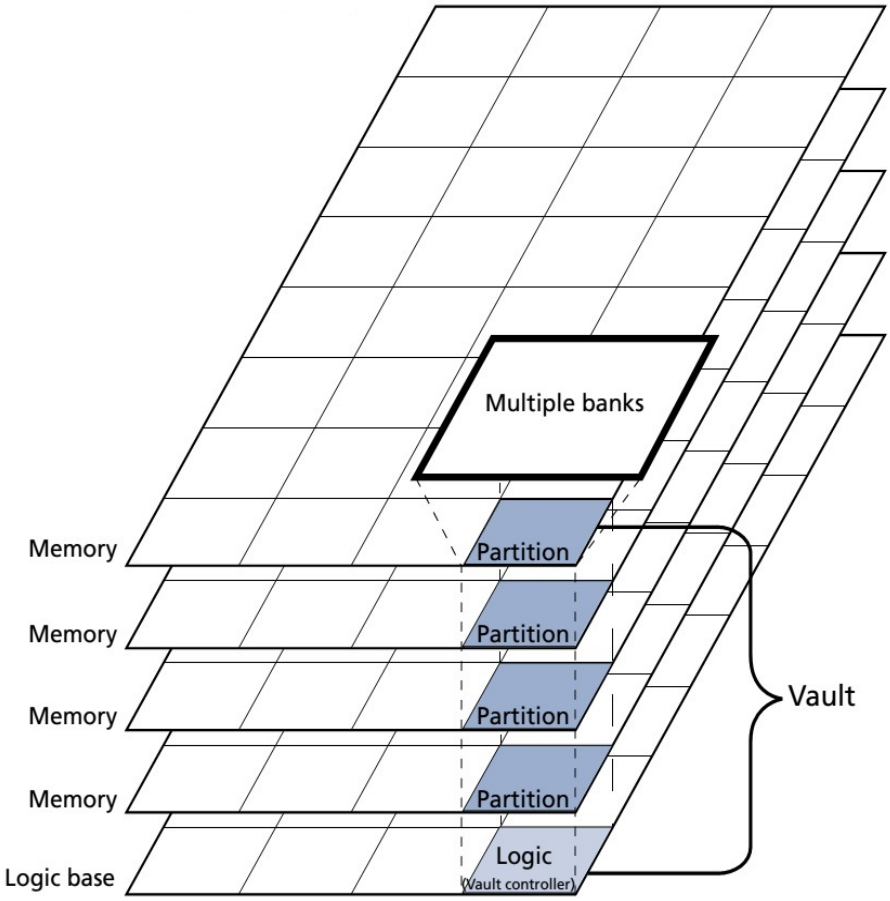


Figure 2.1: HMC Organization

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.

2.2 Logic Base Architecture

The logic base manages multiple functions for the HMC. It has multiple I/O links which are full duplex. The data sent and received through link is serialized. The logic base contains memory controllers for all the vaults. It manages the routing and buffering between I/O links and vaults. It has Mode and Configuration registers. It also contains BIST for the memory and logic layers. It also supports JTAG,I2C,SPI interfaces for maintenance purpose. Below given is the block diagram of 4-link configuration.

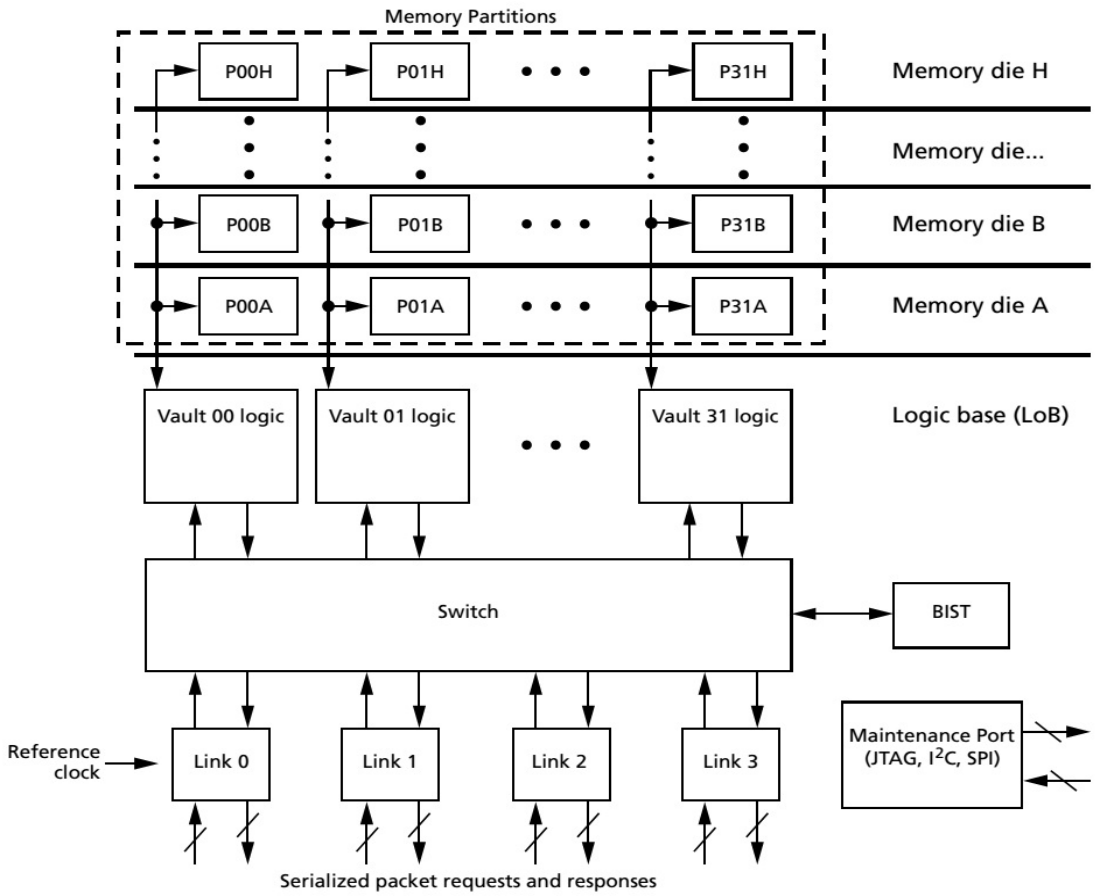


Figure 2.2: HMC Block Diagram

All HMC vaults are connected to external I/O links through crossbar switch as shown in the figure above. External I/O links contain 16 input lanes and 16 output lanes for full duplex operation. This is called full-width configuration. HMC can also support half-width(8 lanes) and quarter-width(4 lanes) configuration. Number of links that HMC can support varies from 2 to 4. Speed of each link varies from 12.5 to 30 Gb/s. The memory

density it can support is either 4 GB or 8 GB. Number of vaults is 32. Memory banks can be either 256(4 GB) or 512(8 GB). Maximum aggregate link bandwidth it can support is 480 GB/s. Maximum suported DRAM data bandwidth is 320 GB/s and the maximum vault bandwidth it can support is 10 GB/s.

The communication across a link is in the form of packets. Packets specify single, complete operations. For example, a WRITE request of 128 bytes of data. No specific timing can be mentioned for a request i.e the amount it takes to serve a request varies. The reasons for such a behavior are given below.

- The vaults reorder their requests to optimize bandwidths and to reduce average latency
- Independent vaults serving their requests independently of other vaults. So the order in which the responses returned may not be the order in which the requests were made.

CHAPTER 3

DATA TRANSMISSION MECHANISM

HMC talks to the outside world only through I/O links. As per the data transmission mechanism followed by HMC, these links send and receive data in the form of packets. All the read/write requests/responses should be in the form of packets before they are sent across. Each packet is further divided into 128-bit flow units called ”FLITs”. These FLITs are serialized, transmitted across the physical lanes of the link, then re-assembled at the receiving end of the link.

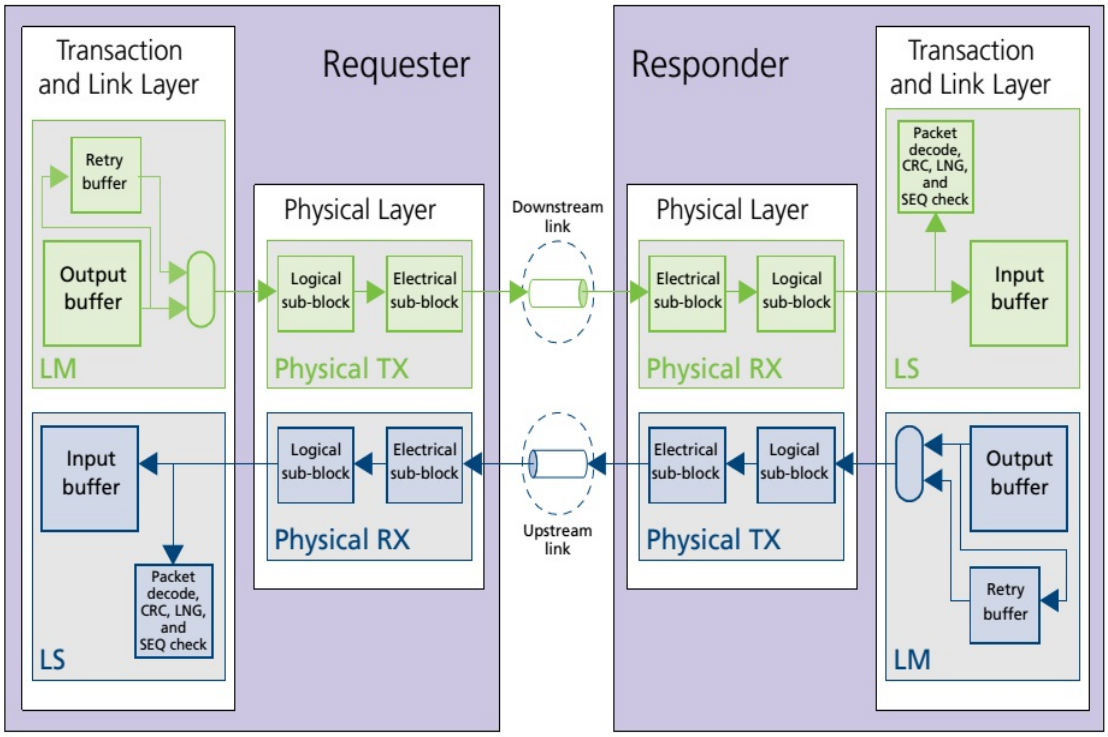


Figure 3.1: Link Data Transmission

As we see in the figure above, two sides of the link are connected to two blocks namely Requester and Responder. Requester is a block which sends requests and receives responses. Responder is a block which sends responses and receives requests. As we can

see here, these blocks can be further divided into 3 layers namely Transaction, Link and Physical layers. Now let us discuss these 3 layers in detail.

3.1 Transaction Layer

Transaction Layer plays a major role in data transmission. This is the layer where the requests/responses get converted to packets and vice-versa. The transaction layer provides the definition of the packets, the fields within the packets, and the packet verification and retry functions of the link. A packet always includes an 8-byte header at the beginning of the packet and an 8-byte tail at the end of the packet. The header includes the command field that identifies the packet type, other control fields, and when required, the addressing information. The tail includes flow and link-retry control fields along with the CRC field. Two different packet types are used at the transaction layer. They are Request and Response packets.

Request Packets These are issued by the requester (host or HMC configured as a pass-thru link). The request packet header includes address information to perform a READ or WRITE operation. Write request packets include data FLITs.

Response Packets These are issued by the responder (HMC configured as a host link). Response headers do not include address information. READ responses include data FLITs, and can optionally include write response tags embedded in the header. Write response packets do not include data FLITs.

Each packet type has its own defined header and tail formats. Packets that consist of multiple FLITs are transmitted across the link with the least significant FLIT number first. The first FLIT includes the header and the least significant bits (LSBs) of the data. Subsequent data FLITs progress with more significant data following. The figure 3.2 shows the layout for packets with and without data.

In the following sections, first we will look at what the request/response packet's Head-/Tail comprises. We will discuss various fields available in Head and Tail of a packet. We

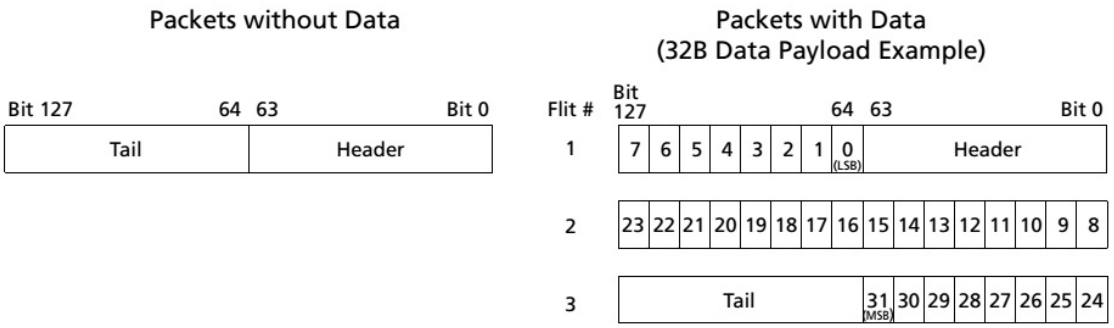


Figure 3.2: Packet Layouts

will also discuss how each field of Head/Tail is useful for packet transmission in a detailed manner.

3.1.1 Packet Formats

Here we see the formats of Request and Response packets. We will see what the Header and Tail of Request/Response packets will consist of.

Request Packets carry request commands from the processor to HMC. The table given below shows the fields of Header of a Request Packet.

Table 3.1: Request Packet Header Fields

Name	**Field Label**	**Bit Count**	**Bit Range**	**Function**
Cube ID	CUB	3	[63:61]	CUB field used to match request with target cube.
Reserved	RES	3	[60:58]	Reserved: These bits are reserved for future address or Cube ID expansion.
Address	ADRS	34	[57:24]	Request address
Reserved	RES	1	[23]	
Tag	TAG	11	[22:12]	Tag number uniquely identifying this request.
Packet Length	LNG	5	[11:7]	Length of packet in FLITs
Command	CMD	7	[6:0]	Packet Command

The table given below shows the fields of Request Packet's Tail.

Table 3.2: Request Packet Tail Fields

Name	**Field Label**	**Bit Count**	**Bit Range**	**Function**
Cyclic Redundancy Check	CRC	32	[63:32]	The error-detecting code field that covers the entir packet.
Return Token Count	RTC	3	[31:29]	Return token count for transaction-layer flow control.
Source Link ID	SLID	3	[28:26]	Used to identify the source link for response routing.
Reserved	RES	5	[25:21]	Reserved
Sequence number	SEQ	3	[20:18]	Incrementing value for each packet transmitted.
Forward Retry Pointer	FRP	9	[17:9]	Retry pointer representing this packet's position in the retry buffer.
Return Retry Pointer	RRP	9	[8:0]	Retry pointer being returned for other side of link

Response packets carry response commands from the HMC to processor. The table given below shows the fields of Header of a Response Packet.

Table 3.3: Response Packet Header Fields

Name	**Field Label**	**Bit Count**	**Bit Range**	**Function**
Cube ID	CUB	3	[63:61]	The target cube inserts its Cube ID number into this field.
Reserved	RES	19	[60:42]	The host will ignore bits in this field.
Source Link ID	SLID	3	[41:39]	Used to identify the source link for response routing.
Reserved	RES	5	[38:34]	The host will ignore bits in this field.
Atomic Flag	AF	1	[33]	Atomic Flag
Reserved	RES	10	[32:23]	The host will ignore bits in this field.
Tag	TAG	11	[22:12]	Tag number uniquely associating this response to a request
Packet Length	LNG	5	[11:7]	Length of packet in FLITs
Command	CMD	7	[6:0]	Packet Command

The table given below shows the fields of Response Packet's Tail.

Table 3.4: Response Packet Tail Fields

Name	**Field Label**	**Bit Count**	**Bit Range**	**Function**
Cyclic Redundancy Check	CRC	32	[63:32]	The error-detecting code field that covers the entire packet.
Return Token Count	RTC	3	[31:29]	Return token count for flow control.
Error Status	ERRSTAT	7	[28:22]	Error status bits
Data Invalid	DINV	1	[21]	Indicates validity of packet payload.
Sequence number	SEQ	3	[20:18]	Incrementing value for each packet transmitted.
Forward Retry Pointer	FRP	9	[17:9]	Retry pointer representing this packet's position in the retry buffer.
Return Retry Pointer	RRP	9	[8:0]	Retry pointer being returned for other side of link.

The tables given above show the various fields available in the packet's header/tail. Below we will see how each field of the packet is used for successful transmission of data between processor and HMC.

3.1.2 Commands

Each request/response packet carries a command associated with it. CMD field of the packet's header indicates specific command associated with the packet. From a command, the following details can be inferred.

- Whether the request is read/write
- length of a packet
- the size of the data associated with a request/response i.e. it tells us no of bytes need to be read/written to/from the memory
- if it is a request packet, whether response needs to be returned or not.

Commands can be divided into 3 categories based on the type of packets they reside in. We will look at them one by one.

3.1.2.1 Request Commands

Request commands are issued by the requester i.e. processor. These are available in request packet headers. Here are the various commands available.

- Write Requests size of the data can vary from 16-byte to 256-bytes
- Posted Write Requests size of the data can vary from 16-byte to 256-bytes
- Read Requests size of the data can vary from 16-byte to 256-bytes
- Mode Read/Write Requests
- Atomic Requests

3.1.2.2 Response Commands

Response Commands are issued by the responder i.e. HMC. These are available in response packet headers. The table below shows various commands available.

Table 3.5: Response Commands

Command Description	**Symbol**	**CMD field**	**Packet Length in FLITs**
READ response	RD_RS	0111000	1 + data FLITs
WRITE response	WR_RS	0111001	1
MODE READ response	MD_RD_RS	0111010	2
MODE WRITE response	MD_WR_RS	0111011	1
ERROR response	ERROR	0111110	1

3.1.2.3 Flow Commands

Flow Commands are issued by link master to facilitate retry and flow control on the link. These are available in flow packet headers.These are not considered as requests so response command is associated with them. The table 3.6 shows various commands available.

Table 3.6: Flow Commands

Command Description	Symbol	CMD field	Packet Length in FLITs
Null	NULL	0000000	1
Retry Pointer Return	PRET	0000001	1
Token Return	TRET	0000010	1
Init Retry	IRTRY	0000011	1

3.1.3 Packet Length

Packet Length is indicated by Length(LNG) field which is available in the packet's header. It indicates the total number of FLITs available in the packet. It's a 5-bit field i.e. the maximum length of a packet is 32. A packet that contains no data FLITs would have LNG = 1; this represents the single FLIT holding the 8-byte header and 8-byte tail. When generating response packets, the necessary data size is determined from the CMD field in the corresponding request packet.

3.1.4 Tags

Each request is tagged with a unique number. Tag of a request resides in the TAG field of packet's header. TAG of a response is also unique as it is copied from the corresponding request packet. Operational closure for requests is accomplished using the tag fields in request and response packets. The tag value remains associated with the request until a response is received indicating that the request has been executed. Tags in READ requests are returned with the respective read data in the read response packet header. Write response tags are returned within a write response packet. In the case of posted write requests, because no response is returned and the HMC does not use the TAG field of posted write requests, the host can populate this field with any value. When multiple host links are connected to the HMC, each response packet will be returned on the same link as its associated request. This keeps the tag range independent for each host link.

Tag fields in packets are eleven bits long, which is enough space for 2048 tags. All tags are available for use. Tag assignment and reassignment are managed by the host. There is no required algorithm to assign tag values to requests. HMC does not use the tag for

internal control or identification, only for copying the tag from the request packet to the response packet.

Tags are assigned by control logic at the host link master and must not be used in another request packet until a response tag with the same tag number is returned to that host link. Other host links will use the same tag range of 2048 tags, but they are uniquely identified by their association with each different host link. Additionally, in a multiHMC topology, the host could choose to expand the number of usable tags by associating each tag set (based on host link) with the target cube of the request. This is enabled by the inclusion of the cube number in each response packet. Thus, the total maximum number of unique tags is 2048 times the number of host links of one HMC, times the number of cubes in the chain. For example, if four links of one HMC are connected from an HMC to the host, there are 8192 usable tags for requests to the HMC.

As an implementation example, a host control logic might decide to provide a time-out capability for every transaction to determine whether a request or response packet has been lost or corrupted, preventing the tag from being returned. The timer in this implementation example would consider link retry periods that occur to account for response packets that are delayed, but still returned. Host timeout values must be determined from performance modeling of the HMC using specific host request streams, with its address patterns and transaction mix, to determine expected maximum latency of transactions. The transaction timeout period must be guard-banded to allow for possible significant variability in latency doe to host request stream variability.

3.1.5 Token counts

Flow control occurs in both directions on each link. The flow of transaction layer packets is managed with token counts. Token counts are related to the available space in the input buffer. It will help the transmitter to know the available space in input buffer of the receiver. Token counts are represented as RTC field available in the tail of a packet. The value of RTC field is not equal to the number of token counts returned. Actually, number of token counts returned is encoded in the RTC field. The encoding is listed in Table 3.7 .

Table 3.7: RTC Encoding

RTC[2:0]	Token Count Returned
000	0
001	1
010	2
011	4
100	8
101	16
110	32
111	64

Each token represents the storage to hold one FLIT (16 bytes). The link slave input buffer temporarily stores transaction layer packets as they are received from the link. (Flow packets are not stored in the input buffer, and therefore, are not subject to flow control.) The minimum size of this buffer must be equal to or greater than the number of FLITs in a single packet of the largest supported length, but in general can hold many FLITS to enable a constant flow across the link. The available space in this input buffer is represented in a token count register at the link master. This gives the link master knowledge of the available buffering at the other end of the link and the ability to evaluate whether there is enough buffer space to receive every FLIT of the next packet to be transmitted.

The token count register is loaded during the initialization sequence with the maximum number of tokens representing the available buffer space at the link slave when it is empty. As the link master sends each transaction layer packet across the link, it must decrement the token count register by the number of FLITs in the transmitted packet. As the packet is received and stored in the input buffer, its FLITs use up the space represented by the decremented value of token count register at the link master. As each packet is read out of the input buffer, it is forwarded to a destination, and its FLIT location is freed up. This requires a mechanism to return packet tokens to the link transmitter, using the opposite link direction.

The input buffer control logic sends a count (representing the number of FLITs that were read out of the input buffer when the packet was forwarded) to the local (adjacent) link master. This count is returned in the return token count (RTC) field of the next possible

packet traveling in the opposite direction on the link. At the link slave, the RTC field is extracted from the incoming packet and sent back to the original link transmitter where its value is added to the current value of the token count register. If no transaction layer packet traffic is occurring in the return direction, the link master must create a flow packet known as a token return (TRET) to return the token. Not returning tokens can lead to performance degradation or stalling. TRET is described in TOKEN RETURN (TRET) Command.

Appropriate sizing of the link slave input buffer requires weighing the trade-offs between latency, throughput, silicon real estate, and routability. If there is a temporary pause in the packet forwarding priority at the output of the input buffer (internal switch conflicts, or destination vault flow control), a packet might not be emptied from the input buffer. This would cause a pause in the return of the tokens. As long as this is infrequent and the input buffer is sized to accommodate it, the packet flow will not be disrupted unless the link is running at 100 percent busy. If the input buffer is empty, a specific implementation may choose to bypass the input buffer and forward a packet immediately to its destination. In this case, the tokens for the packet would be immediately returned to the link master.

As noted previously, the link master must not transmit a packet if there is insufficient space in the input buffer at the other end of the link. The LNG field within the header of each outgoing packet must be compared to the token count register to determine whether the packet should be transmitted or the packet stream should be paused.

3.1.6 Error Detection

Cyclic Redundancy Check(CRC) is used as an error detection method in this protocol. Calculated CRC for the entire packet is included in the CRC field of a packet's tail. It covers header, all data and non-CRC tail bits. CRC will be generated again at the receiving end of the link. Error is detected if the generated CRC does not match the CRC embedded in the packet. If there is an error, the packet will be re-transmitted.

The CRC algorithm used on the HMC is the Koopman CRC-32K. This algorithm was chosen for the HMC because of its balance of coverage and ease of implementation. The

polynomial for this algorithm is:

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

The CRC calculation operates on the LSB of the packet first. The packet CRC calculation must insert 0s in place of the 32-bits representing the CRC field before generating or checking the CRC. For example, when generating CRC for a packet, bits [63: 32] of the Tail presented to the CRC generator should be all zeros. The output of the CRC generator will have a 32-bit CRC value that will then be inserted in bits [63:32] of the Tail before forwarding that FLIT of the packet. When checking CRC for a packet, the CRC field should be removed from bits [63:32] of the Tail and replaced with 32-bits of zeros, then presented to the CRC checker. The output of the CRC checker will have a 32-bit CRC value that can be compared with the CRC value that was removed from the tail. If the two compare, the CRC check indicates no bit failures within the packet.

CRC field in the tail of every packet maintains the packet integrity. Because the entire packet (including header and tail) is transmitted from the source link all of the way to the destination vault, CRC is used to detect failures that occur not only on transmission across the link, but along the entire path. CRC may be regenerated along the path if flow control fields within the header or tail change. CRC regeneration is done in an overlapped fashion with respect to a CRC check to ensure that no single point of failure will go undetected.

There may be cases when the first FLITs of a packet are forwarded before the tail is received and the CRC is checked. This occurs in the HMCs 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. A successful link retry will result in the original packet being resent, thus replacing the poisoned packet.

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 will receive the poisoned packet and must drop it. The vault controller will retry the DRAM request, and upon a successful DRAM access, another response packet will be generated to replace the poisoned one.

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, and ERRSTAT fields are not valid, but the flow control fields (FRP, RRP, RTC) are valid.

3.2 Link Layer

The link layer provides the low-level handling of the packets at each end of the link. This layer is not assigned much of a task apart from handling communication across the link not associated with the transaction layer. This consists of the transmission of NULL FLITs and flow packets. These packets actually are dropped or created only at Link layer.

NULL FLITs All FLITs of a packet must be transmitted sequentially, without interruption, across the link. NULL FLITs are generated at the link layer when no other packets are being transmitted. A NULL FLIT is an all-zeros pattern that (in common with all FLITs) is scrambled prior to transmission. Any number of packets can be streamed back-to-back across the link, or they can be separated by NULL FLITs, depending upon system traffic and transaction layer flow control. There is no minimum or maximum requirement associ-

ated with the number of NULL FLITs transmitted between packets. NULL FLITs are not subject to flow control. The first nonzero FLIT following a NULL FLIT is considered to be necessarily the first FLIT of a packet. Data FLITs within a packet may be all zeros, but these lie between a header FLIT and a tail FLIT and therefore cannot be misinterpreted as NULL FLITs.

Flow Packets Flow packets are generated by the link master to pass flow control and retry control commands to the opposite side of the link. Flow packets are sent when no other link traffic is occurring or when a link retry sequence is to be initiated. Flow packets are single-FLIT packets with no data payload and are not subject to flow control. Flow packets utilize the same header and tail format as request packets, but are not considered requests, and do not have corresponding response commands.

3.3 Physcial Layer

Physical layer handles serialization, transmission, and deserialization. This layer takes the FLITs sent from link/transaction layer and sends them over the outgoing lanes. It also receives the data from the incoming lanes and sends the FLITs created from the data to the link/transaction layer. It consists of two sub blocks namely logical sub-block and electrical sub-block. Here we discuss the logical sub-block.

3.3.1 Logical Sub-Block

The transmitting logical sub-block contains serializer and scrambler, while the receiving logical sub-block consists of deserializer and descrambler. Each FLIT takes the following path.

1. 128-bit FLIT is sent over a wire(128-bit width) from link layer to the transmitting logical sub-block in the physical layer.
2. The transmitting logical sub-block serializes each FLIT and drives it across the link interface in a bit-serial form on each of the lanes.
3. The receiving logical sub-block deserializes each lane and recreates the 128-bit parallel FLIT. Receive deserializer FLIT alignment is achieved during link initialization.

4. The 128-bit parallel FLIT is sent to the link layer.

The following subsections will describe the serialization and scrambling in detail.

3.3.1.1 Serialization and Deserialization

Link serialization occurs with the least-significant portion of the FLIT traversing across the lanes of the link first. During one unit interval (UI) a single bit is transferred across each lane of the link. For the full-width configuration, 16 bits are transferred simultaneously during the UI, so it takes 8UIs to transfer the entire 128-bit FLIT. For the half-width configuration, 8 bits are transferred simultaneously, taking 16UIs to transfer a single FLIT. The following table shows the relationship of the FLIT bit positions to the lanes during each UI for full-width configuration.

Table 3.8: Unit Interval FLIT Bit Positions for Full-Width Configuration

	Lane															
UI	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
2	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
3	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
4	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65	64
5	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81	80
6	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97	96
7	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113	112

Deserialization The receiving sub-block is connected to 16 incoming lanes for a full-width configuration. It receives the data of 16 bits in each UI as shown in the table above. Once 8 UIs are completed, the FLIT is built and sent to the link layer.

3.3.1.2 Scrambler and Descrambler

The protocol implements a scrambler to provide serial data with adequate edge density for AC coupling and data driven clock recovery. Both scrambler and descrambler contain the same logic. The difference is, scrambler lies in the transmitting section and the descrambler resides in receiving logical sub-block. Figure 3.3 shows how they are included in the path of data flow.

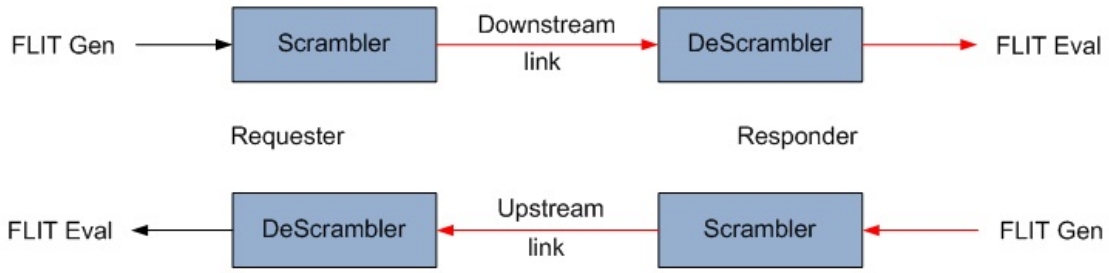


Figure 3.3: Scrambler and Descrambler paths

For scrambling and descrambling we use Linear Feedback Shift Register(LFSR). Polynomial of LFSR is set to $1 + x^{-14} + x^{-15}$. For scrambling, we XOR the data with LSB of LFSR. We need a LFSR for each lane. So there will be 16 LFSRs in each of Scrambler and Descrambler for full-width configuration. Seeding of LFSRs vary per lane. Below given are the seeding values per lane.

Table 3.9: Scrambler Seed Values

Lane	**Seed Value**
0	15'h4D56
1	15'h47FF
2	15'h75B8
3	15'h1E18
4	15'h2E10
5	15'h3EB2
6	15'h4302
7	15'h1380
8	15'h3EB3
9	15'h2769
10	15'h4580
11	15'h5665
12	15'h6318
13	15'h6014
14	15'h077B
15	15'h261F

3.4 Retry Mechanism

As the link transmits data at a very high speed, we need a mechanism to take care of correcting transmission failures to increase system reliability and availability. Link retry is

one such fault-tolerant feature that recovers the link when link errors occur. We rely upon CRC, SEQ checks to detect transmission errors on link. Here is the mechanism to correct the transmission failures.

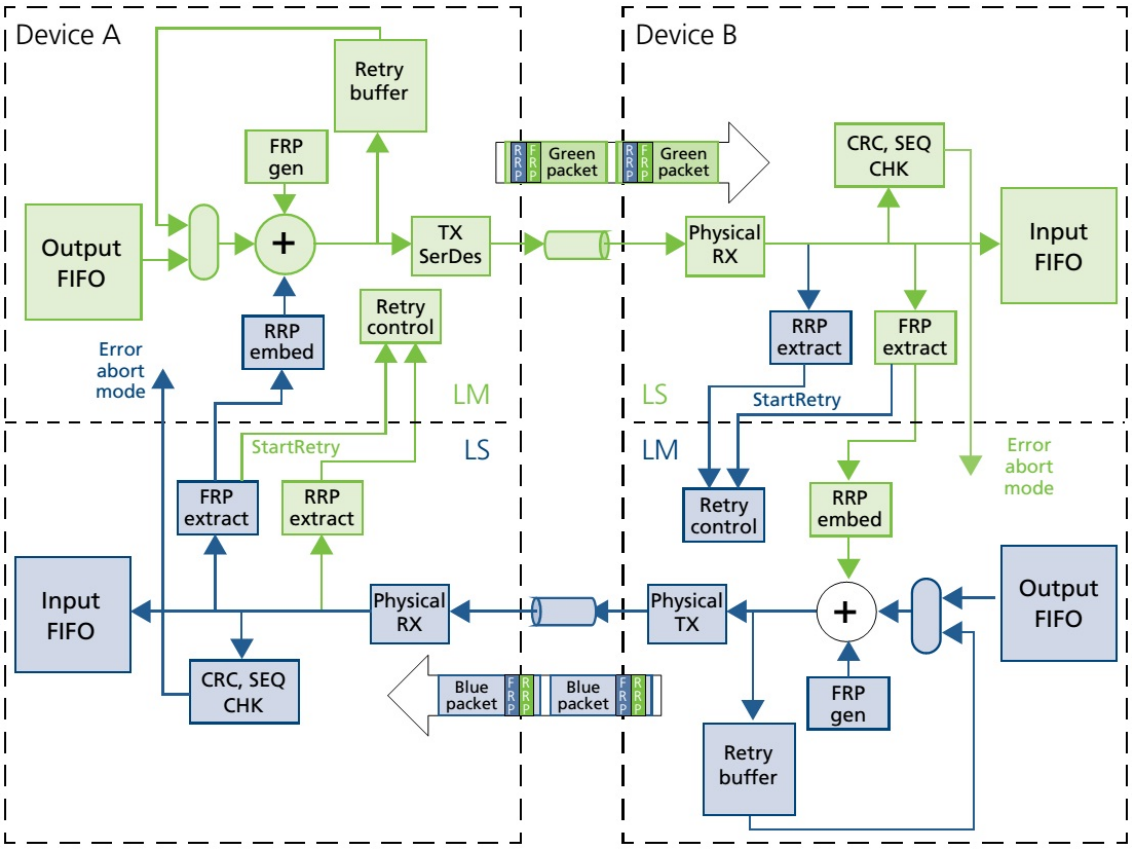


Figure 3.4: Retry Block Diagram

In link master, we maintain a Retry Buffer. Link master keeps a copy of each packet transmitted across the link in the Retry Buffer until it gets acknowledgement saying the packet is transmitted without error. This acknowledgement process happens through Retry Pointers embedded in the tail of the packet. Retry Pointer points to the position of the packet in the Retry buffer. Retry Pointer is sent as FRP in tail along with the packet. If the packet is transmitted successfully, it will sent as RRP in tail of some other packet traveling in the opposite direction.

If there is an error(CRC,SEQ) which is detected by link slave, then link slave enters error abort mode. In this mode, it will ask the local link master to send a StartRetry flag back to the transmitting end using the other side of the link. The whole thing is captured

in the figure 3.4 . The information related to the packets(FRP,RRP,buffer) that are flowing from Device A to B are colored in Green, where as the information relating to the packets traveling in opposite direction are colored in Blue.

3.4.1 Retry Pointers

As mentioned earlier, the retry pointer represents the packet's position in the retry buffer. The retry pointer transmitted with the packet points to the retry buffer location + 1 (to which the tail is being written). This is the starting address of the next packet to be transmitted.

There are two versions for the retry pointer.

FRP Retry pointer will be in the form of FRP as it is travelling in the forward direction, this will be along with the packet. FRP is sent with every packet that is part of the normal packet flow. This is embedded in FRP field of in the packet's tail.

RRP RRP is a copy of FRP in the opposite direction of the link. This is embedded in the RRP field of any packet's tail that is traveling in the opposite direction. If there is no traffic flowing in the opposite direction, then a PRET packet is created to send the FRP received as RRP to the other side of the link.

The figure 3.5 shows implementation of Retry Buffer. The retry buffer is addressed using FLIT addresses, and the retry pointer represents a FLIT position in the retry buffer. Retry pointer of a packet always points to header FLIT of next packet in the Retry Buffer. There will be two pointers to track the status of FLITs stored in the buffer. Read pointer points to the FLIT that is going to retire from the buffer next. Write pointer points to the position where next FLIT is going to be written. As retry buffer is a circular one, these pointers wrap around the positions in the buffer.

Write pointer is incremented as the FLITs are written into the retry buffer from Output buffer. When there is no error, read pointer will be changed to received RRP. That means, it is retiring all the FLITs of the packet which transmitted without errors. If there is an

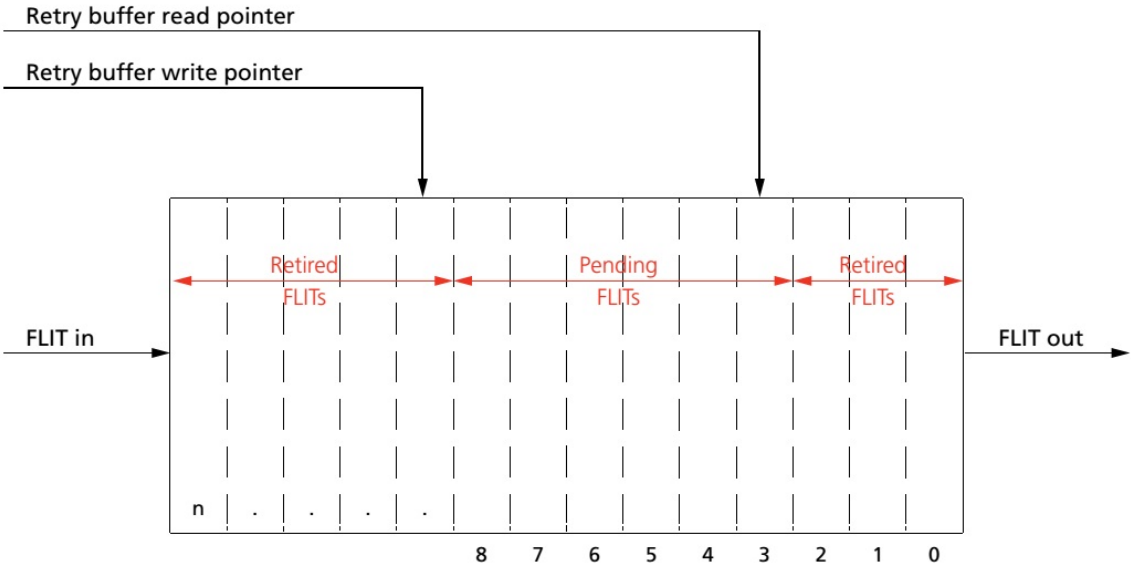


Figure 3.5: Retry Buffer Implementation

error, link master starts a retry sequence. During retry sequence, FLITs will be read out from buffer one by one and read pointer is incremented. FLITs will be retired until the read pointer equals the write pointer.

CHAPTER 4

IMPLEMENTATION DETAILS

This chapter describes how the interface block is implemented in Bluespec System Verilog. Then we discuss how the working of the block is verified by writing a Test Bench.

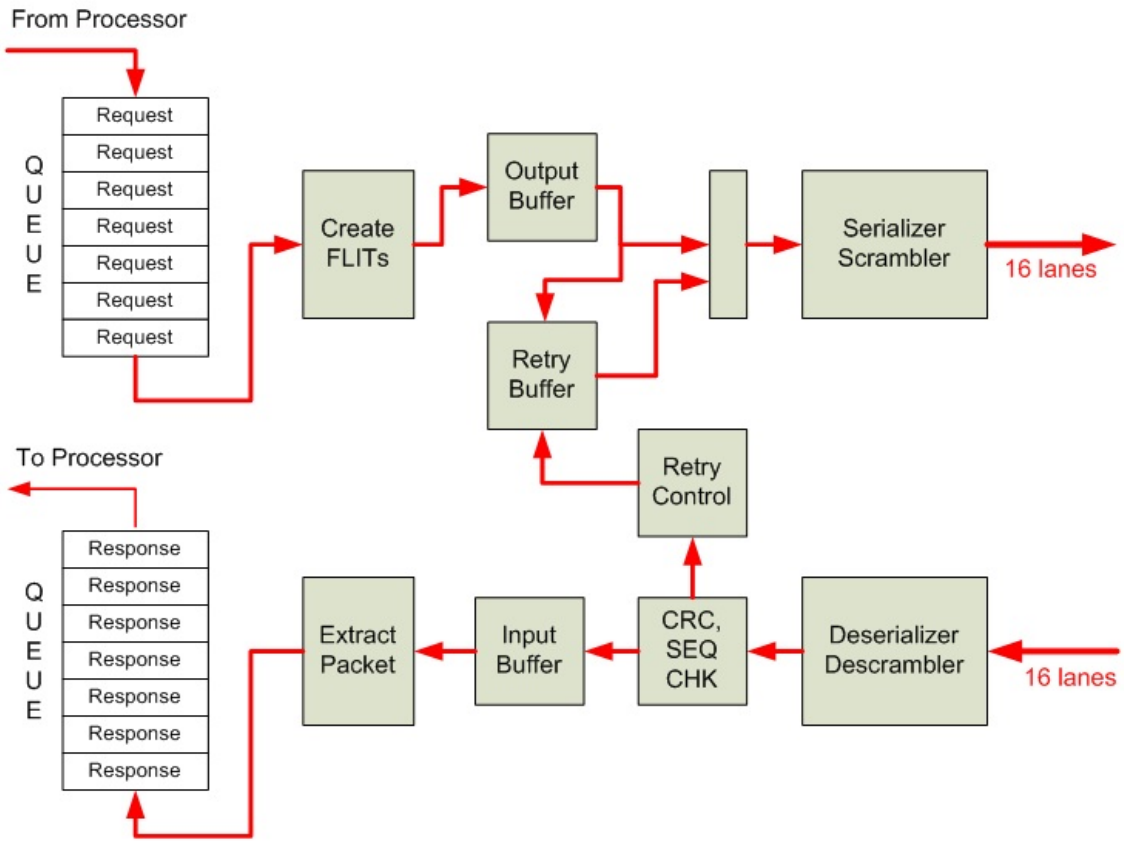


Figure 4.1: Requester Block

4.1 Design

The interface block being on the processor side can be called as Requester as it sends requests to the Memory. Figure 4.1 shows the block diagram of the Requester block.

For easier understanding, the interface block can be divided into three parts namely sending part, receiving part and retry section. We will discus in detail in the following sections.

4.1.1 Sending

Sending part will handle the requests sent from processor. It will convert the requests into packets and sends them to Memory. The packets are divided into FLITs. FLITs are serialized and scrambled before they sent over the link. Here are microscopic details of the Sending part.

Requests (read, write) sent to the Memory from the processor are added to Request Queue. Requests taken out of Queue one by one will be sent to CreateFLITs module. CreateFLITs module creates the packet for the corresponding request. The packet will be created in the form of FLITs. CRC for a packet will be generated before the FLITs are added to Output FIFO Buffer. Each FLIT taken out from Output FIFO will be sent to Serializer block. Serializer block serializes the 128-bit FLIT into 16-bit subFLITs. That means, each FLIT will be transformed into 8 subFLITs over 8 Unit Intervals of time. Each subFLIT is converted to 16 1-bit lines. Each line will be scrambled before it is connected to 16 sending lanes of link. Below we discuss each block in detail.

4.1.1.1 Request Queue

Request Queue is a FIFO data structure, an instantiation of mkSizedFIFO. The size of Queue is fixed as 20 for testing purpose. i.e. It can hold a maximum of 20 Requests. Each Request contains request command, address, data to be written to the memory if it is a write request.

4.1.1.2 CreateFLITs

Each request taken out of Request Queue is stored in a PipelineFIFO. The current request from PipelineFIFO will be dequeued and enqueued with a new request once the packet for the current request is generated and sent to Output FIFO. Number of FLITs generated for

a request depends on the Request command available in the request. The block diagram of the CreateFLITs is given below.

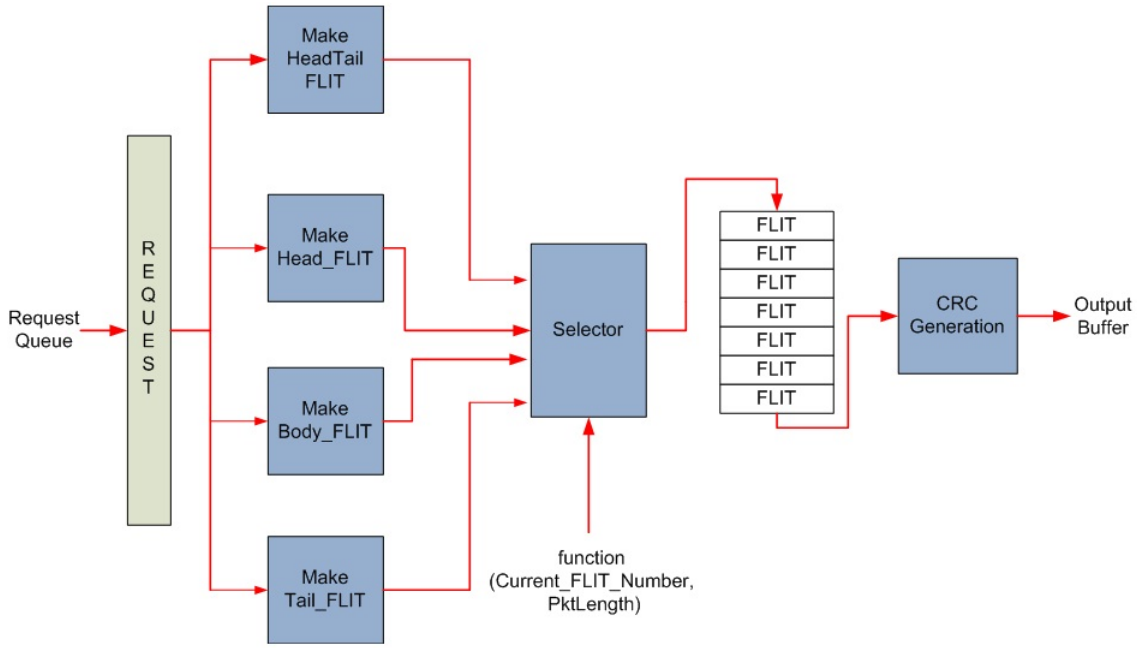


Figure 4.2: CreateFLITs Block

Each FLIT will go through CRC Generator block before getting added to Output FIFO. The Tail FLIT of a packet will be embedded with the CRC generated for the entire packet before sending it to Output FIFO. When generating FLITs, we will be generating 4 types of FLITs namely Head FLIT, Tail FLIT, Body FLIT and HeadTail FLIT. Each packet may contain one or many of them depending on request command. If the number of FLITs is 1, then a packet will consists of only one HeadTail FLIT. If the number of FLITs is 2, then the packet will consists of one Head FLIT and one Tail FLIT. If the number of FLITs is more than 2, apart from one Head FLIT and one Tail FLIT the packet will contain one or more Body FLITs. The FLITs will be generated in the following order and sent to CRC Generation block. Head FLIT first, then Body FLITs and then Tail FLIT. Now lets have a closer look at the CRC Generation block in details.

CRC Generation 32-bit CRC is generated using Koopman32K algorithm. CRC Generator contains a Register for the FLIT it is processing. The CRC Generator contains a MyCRC module which takes 32 bits of data and results 32-bit CRC in one clock cycle. So

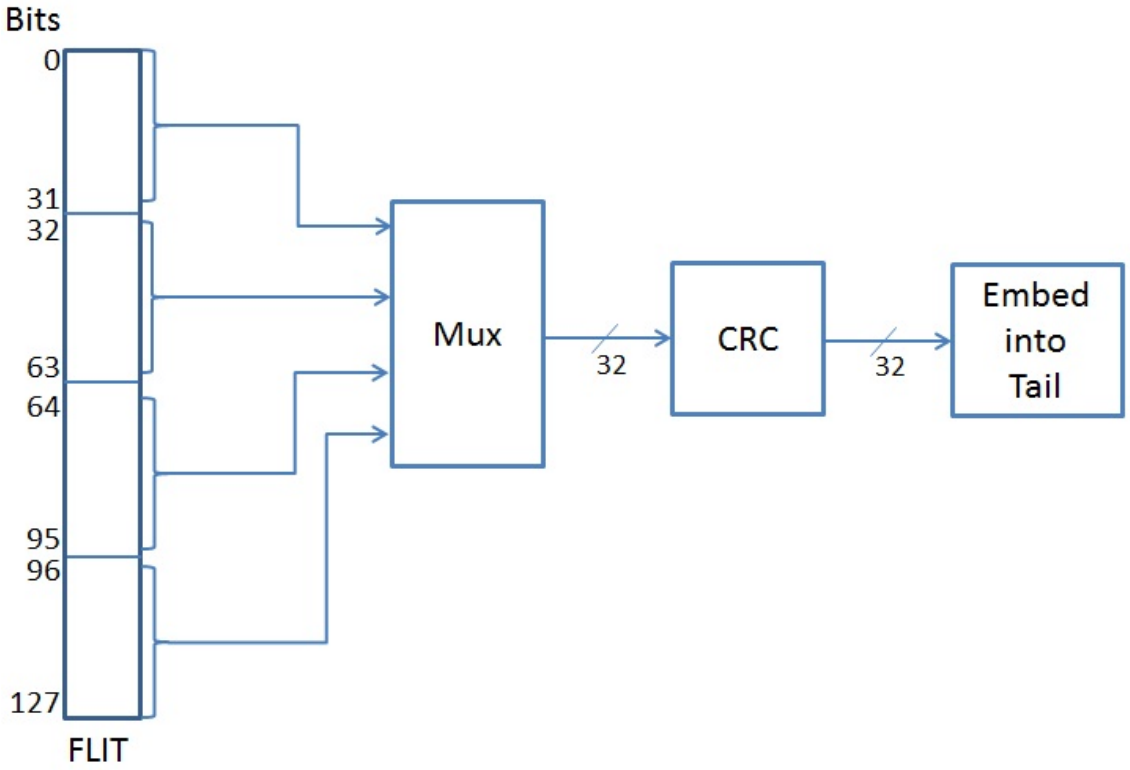


Figure 4.3: CRC Generation Block

we need 4 clock cycles to processor a single FLIT. Thats why we need a Register to store the FLIT for 4 clock cycles. We process Lower Significant bits of a FLIT first, then come the Higher Significant bits. At the end of the 4th clock cycle FLIT will be transferred to Output FIFO if it is not a Tail or HeadTail FLIT. If it is a Tail or HeadTail FLIT, we need to wait for one more cycle for CRC result before we embed that into Tail of the packet. So for Head and Body FLITs, CRC generator takes 4 clock cycles to process where as it takes 5 clock cycles for Tail and HeadTail FLITs. The block diagram of the CRC Generation is given in figure 4.3 .

4.1.1.3 Output Buffer

This is a FIFO data structure, an instantiation of mkSizedFIFO. Outgoing FLITs will be stored in this Buffer. Size of buffer is 20, i.e we can store a maximum of 20 FLITs. If the FIFO is full, we will have a stall in the pipeline.

4.1.1.4 Serializer and Scrambler

128-bit parallel FLIT sent from Output Buffer is first serialized and then scrambled.

Serializer FLITs taken from Output FIFO or Retry Buffer are sent to Serializer. 128-bit FLIT is saved in a register. 16-bit wire rolls over the 128-bit FLIT register connecting 16 bits of the FLIT every cycle from LSB to MSB. It will take 8 clock cycles to transfer a FLIT on to 16-bit wire. This 16-bit wire is connected to 16 1-bit wires.The block diagram of Serializer is given below.

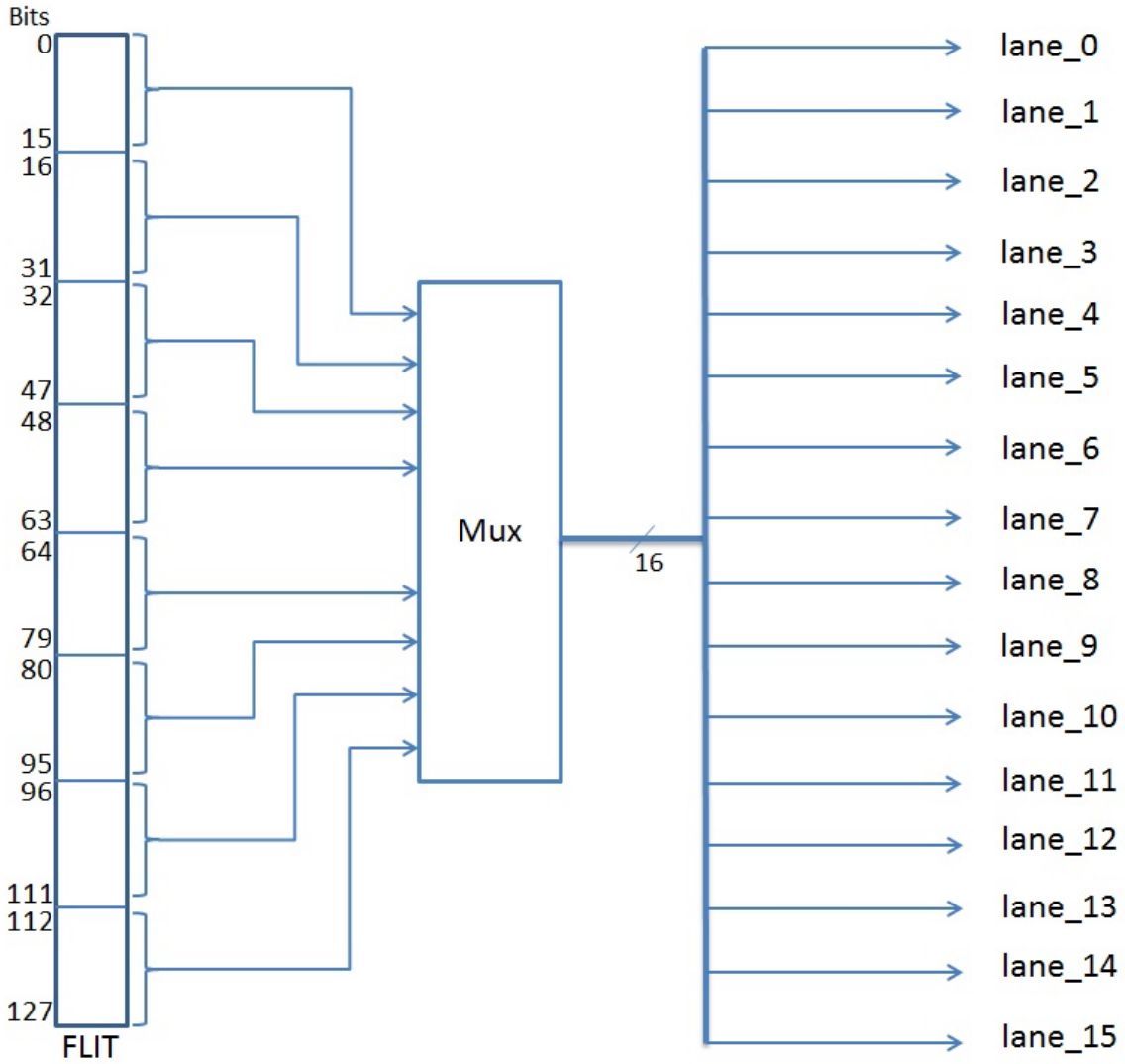


Figure 4.4: Serializer Block

Scrambler There are 16 15-bit LFSRs created to scramble the data on 16 wires. Each LFSR is seeded to a unique value as per the specifications. Each 1-bit wire is XORed with LSB of LFSR and the result is connected to outgoing lane. Thus, 16 1-bit wires are XORed with 16 LFSR LSBs and connected to 16 outgoing lanes. Below is the figure of Scrambler.

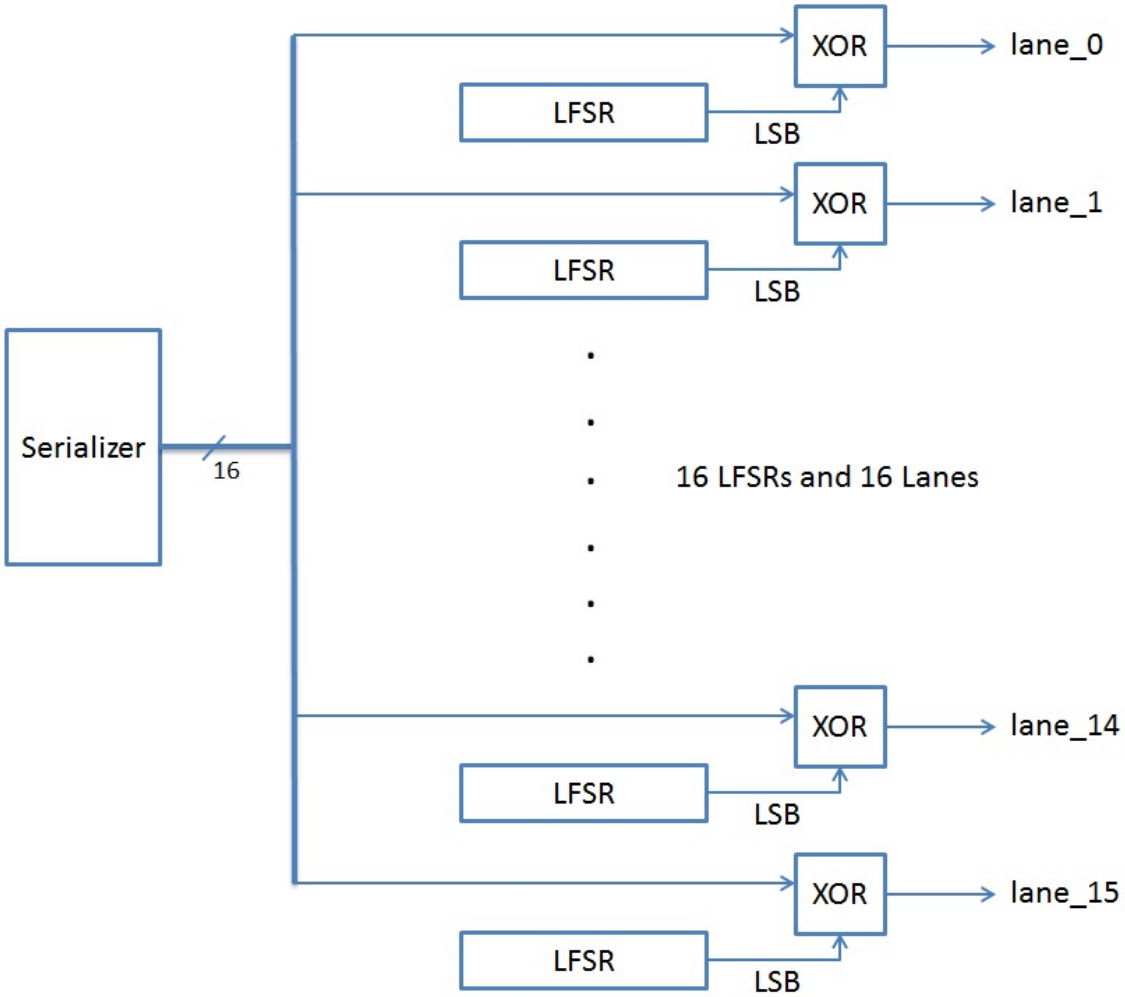


Figure 4.5: Scrambler Block

4.1.2 Receiving

Receiving part will handle the response packets received from Memory. Responses will be extracted from the packets received and sent to Response Queue. Here are the microscopic details of Receiving part.

Responses from the Memory are received in the form of FLITs. 16 Receiving lanes will be descrambled before they are connected to 16 1-bit lanes. These 16 1-bit lanes are grouped to form a 16-bit subFLIT. SubFLITs received over the 8 clock cycles are grouped together to form a 128-bit FLIT. Each FLIT will be marked as either Head/Tail/Body FLIT before they are sent for CRC checking. After CRC checking each FLIT will loaded into Input FIFO Buffer. Each FLIT taken from Input FIFO will be sent for packet extraction module to extract the data or write acknowledgement received from memory. Each response will be added to Response Queue from which processor can get the data.

4.1.2.1 Deserializer and Descrambler

16 incoming lanes are first converted to 16 1-bit wires. In this block, we have 16 15-bit LFSRs to descramble the data. Like in Scrambler, here we XOR 16 1-bit wires with LSBs of 16 LFSRs to get a sub FLIT of size 16 bits. Every cycle we will get a new sub-FLIT. The sub-FLITs collected over 8 cycles are grouped together to form a FLIT. First we get lower significant bits of FLIT. Here we use a simple shifter and OR functionality to create a FLIT with the data collected over the past 8 cycles. Block diagram of Deserializer is given in figure 4.6 .

CRC Checker 128-bit data received from deserializer is tagged as one of the types namely Head, Tail, Body and HeadTail FLIT. Each FLIT goes through CRC checker before gets added to Input Buffer. CRC checker works same way as CRC generator with only one difference. CRC checker not only generates the CRC, it also compares the generated CRC with the CRC embedded in the packet and results error if there is a mismatch. This error will be used to force retransmission of corresponding packet.

4.1.2.2 Input Buffer

This is a FIFO data structure, an instantiation of mkSizedFIFO. Incoming FLITs will be stored in this Buffer. Size of buffer is 20, i.e we can store a maximum of 20 FLITs. If the FIFO is full, we will have a stall in the pipeline.

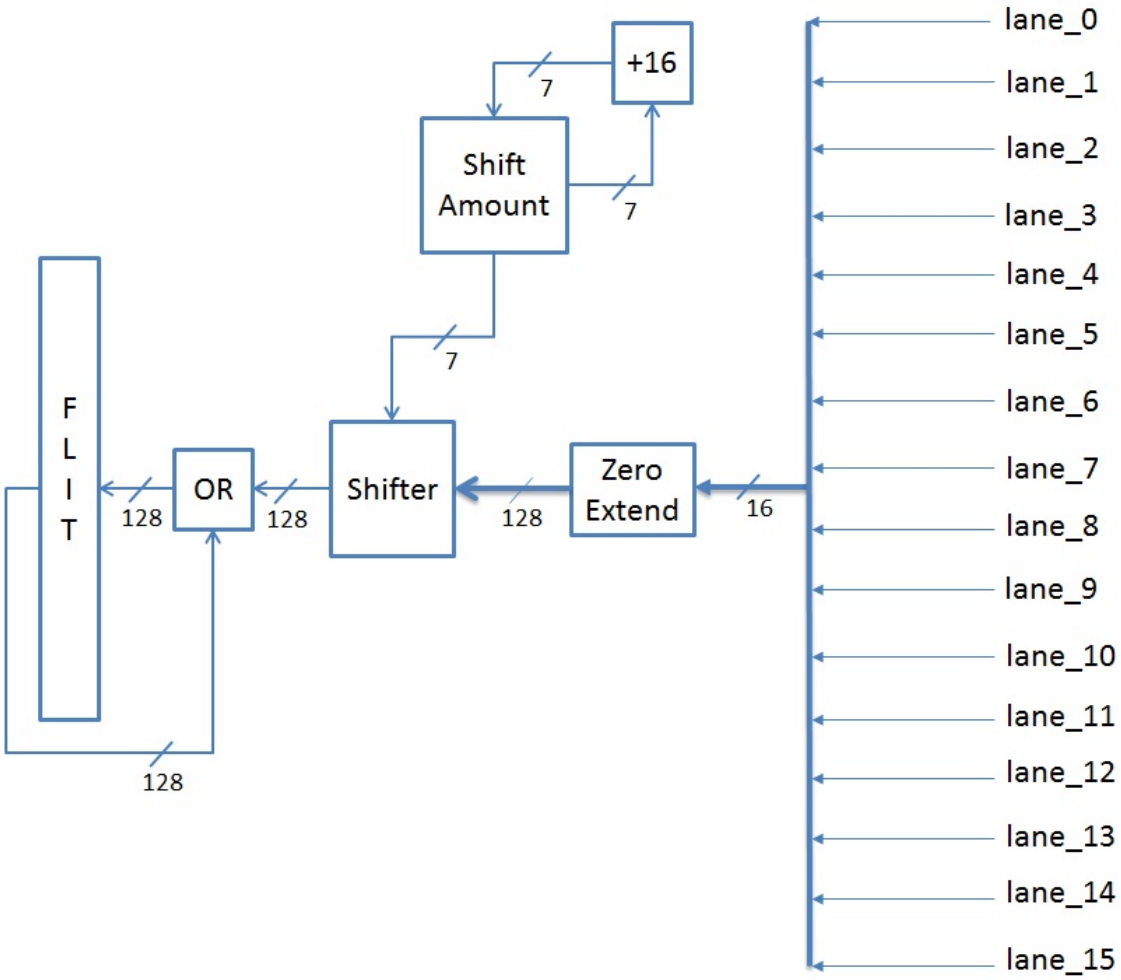


Figure 4.6: Deserializer Block

4.1.2.3 Extract Packet

Each FLIT taken out from Input FIFO will be sent for Packet Extraction. In this module, from each FLIT we take the necessary data to get complete response from a packet. If the packet consists of HeadTail FLIT, it must be a write response packet as there is no data payload. We just take the TAG and errorstat is associated with the packet. Zero errorstat field means the write is successful. If packet consists of more than 1 FLIT, obviously it would be Read Response packet. We extract data payload from the packet by grouping together the data received from Various FLITs. We reject the packets that show CRC errors.

Each Response data taken from the Packet Extraction block will be sent to Response Queue. Output of Response Queue will available for the processor. Response will contain

Tag value of the Request to which it is attached, Errorstat message and the data read from memory if it is a read response.

4.1.3 Retry

Retry mechanism involves majorly Retry Buffer and Retry block.

4.1.3.1 Retry Buffer

Retry Buffer is implemented using a Register File. Two pointers are created namely Read and Write pointers. Read pointer is modified in 2 scenarios.

- Read pointer will be incremented by one when a FLIT is read out from the buffer. This happens when there is a Link Retry.
- Read pointer can also be changed to a specific value. This happens when RRP is received by link slave. Read pointer will be set to the RRP received.

Write pointer is incremented by one whenever there is a FLIT written to the buffer. As Retry buffer needs to be a circular buffer, whenever the pointers are changed it is made sure that they are wrapped around the buffer. Retry Buffer size is randomly selected such that it is adequate enough for Link Retry testing. In real scenarios, Retry Buffer size needs to be calculated based on traffic flow and transmission failures.

4.1.3.2 Retry Control Block

Retry block manages the Retry mechanism of the link. CRCSEQ_Checker sends RRPs extracted from received packets to Retry Control Block. It takes the RRPs and changes the Read Pointer of the Retry Buffer. It also receives error abort signal from CRCSEQ_Checker when there is an error in received packet. It starts the Link Retry process. It manages the state of link master during the Retry mechanism. It maintains the counters related to IRTRY, clearErrorAbort and Retry Timer. It also keeps the FRP of last successfully received packet. This will be used for link retry initiation. When there is a Link Retry, it will prevent Output Buffer from sending out FLITs until Link Retry is finished. When there is a Link Retry, it will make Retry Buffer to send out its FLITs one by one until Read pointer equals Write pointer.

Code The entire code of Requester block is distributed over the following files.

- Requester.bsv
- CreateFLITs.bsv
- Scrambler_Serializer.bsv
- Descrambler_Deserializer.bsv
- Extract_Packet.bsv
- CRCGeneration.bsv
- CRCChecker.bsv
- Typedfn.bsv
- RetryBuffer.bsv
- RetryControlBlock.bsv

4.2 Verification

Verification of the Requester module(interface block) can be done in two phases.
Phase-I: This involves testing the module using simulations on Bluesim and GtkWave
Phase-II: Synthesize and dump the module on to FPGA and connect it to HMC. Altera has some FPGA boards which have HMC installed on them. We can synthesize on to them and test the module.

Phase-II was not carried out, as we couldnt get the FPGA board. So, only Phase-I is carried out and below we see details of Phase-I.

Setup: To test the module on Bluesim, a verification setup has to be created which involves Responder module, a Register File and a test bench along with Requester module. All these units are connected as shown in the figure 4.7 . Here the Responder acts as responder link of HMC and Register File is used in place of HMC. Here the requests sent will have addresses of Register File.

Here are various test cases to verify the working of the module. They are named based on what they are trying to verify.

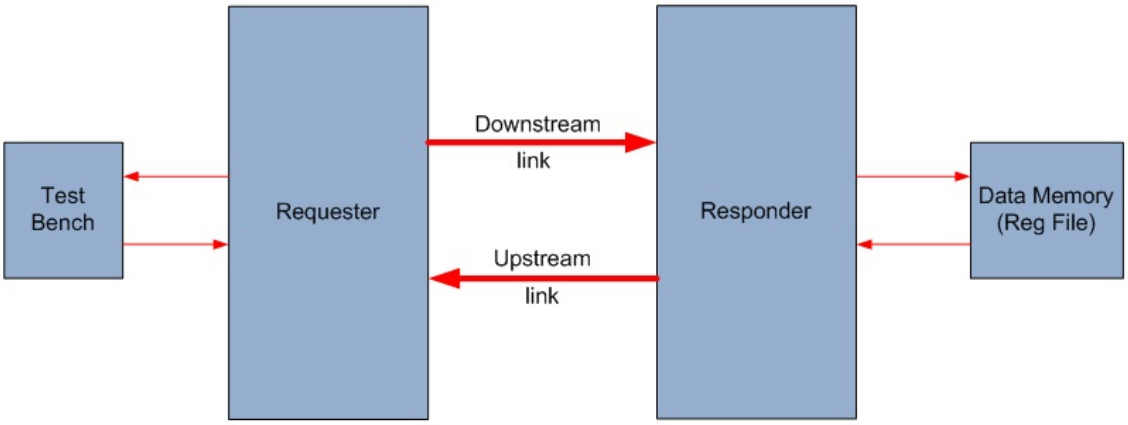


Figure 4.7: Verification Setup

Data transmission: Here the test bench takes the requests from an input file and sends it to the Requester and prints the responses in to an output file. Best way to check the working of interface is to read the data that is written to a particular address i.e. first we send a write request to a particular address and then a read request to the same address. The read response should give us the data that we have to written to the address by write request. This, we can do for various addresses. If data received and data sent matches for all the addresses, then we can say data is transmitted correctly over the link. Mismatch will result in debugging and rectifying the flaw in the design.

Here we have to check whether data is transmitted properly in both directions. We have to check whether requests sent from Test Bench are received correctly by the DataMemory. Similarly, we can check whether responses sent from DataMemory are received correctly by the Test Bench. The requests/responses are converted to various formats such as FLITs, Serialized bits etc. as they go over the link. We can tap at various places in packet flowing path to check whether

- the FLITs are generated correctly for a request/response packet
- the FLITs are serialized and scrambled correctly
- the data is sent properly over the lanes,
- the data is received correctly over the lanes,
- the FLITs are reconstructed correctly
- the requests/responses are properly extracted at the other side of the link

This tapping can be done in 2 ways either by printing to the shell or checking the waveforms on gtkWave. We can also check the contents of all the buffers namely Output, Input and Retry Buffers in every cycle. Location of the error is easily identified by tapping which will be helpful in debugging.

CRC, SEQ check: Sequence checker can be tested by inducting a wrong sequence number while sending a packet from Responder. Same way we can alter some bits of the packet intentionally to test whether the CRC checker is detecting the errors properly.

Retry Mechanism: Once there is an error in the packet transmitted, link slave goes into error abort mode and erroneous packet will be retried. Here, we intentionally alter some bits of the packet which results in CRC error. CRC error will trigger a link retry. We can check the transmissions over the link once the retry starts up on bluesim shell or gtkWave. We can see whether the transmissions over the link are following the Link Retry protocol properly. Once the rety is completed, we have to receive the packet without any errors.

CHAPTER 5

CONCLUSION AND FUTURE WORK

Hybrid Memory Cube specification 2.0 is followed to design the interface module. The work mainly focused on the functionality of the interface module. Functionality wise, the design is working correctly. Performance needs to be taken care in the future revisions. Request template is assumed. Request template needs to be changed when it is integrated with the processor. Sizes of the buffers used are some random numbers that are adequate enough for testing purposes. These buffer sizes need to be modified based on the packet traffic to avoid stalling. At the same time, storage requirement also need to be taken care of while choosing the sizes.

This design works well with a single processor. To work in a multi-core environment this needs to be modified if it has to work like a switch. They are many topologies available for a multi-core environment. The best suited topology must be chosen to meet timing and area requirements. Electrical interface to the module is not implemented as there are no specifications related to it in the HMC spec document. While implementing scrambler and descrambler, PRBS15 1.0 revision is followed instead of PRBS31 2.0 revision, as the seeding values are not mentioned for PRBS31 2.0 in spec document. The module is simulated on Bluesim and the waveforms are checked in GtkWave. As the HMC installed FPGA board is not available, the design is not tested on FPGA.

Future Work:

- Integration with processor
- Optimization in terms of performance and storage
- Synthesis and Test on HMC installed FPGAs
- Implementation of Electrical Interface
- ASIC synthesis to know hardware parameters such Power, Timing and Area

REFERENCES

[1] Hybrid Memory Cube Consortium. ”Hybrid Memory Cube Specification 2.0.” (2014).

[2] Bluespec, Inc, *Bluespec System Verilog Reference Guide,* revision: 17 ed., 2012.

[3] Rishiyur S. Nikhil and Kathy Czeck, *BSV by Example.* Bluespec, Inc, 2010.

APPENDIX A

Bluespec System Verilog

BSV is a language used in design of electronic systems (ASICs, FPGAs and systems). It is a very high level hardware description language and the code written in Bluespec is completely synthesizable to hardware. Because of its high level and completely synthesizable features, it has made many activities that are done in software simulation move to FPGA based simulation.

A.1 Key Features of BSV

- High level atomic rules in place of Verilog method of always block.
- High level Interfaces instead of Verilog method of port list.
- Nested, parameterizable interfaces, allowing easy construction of complex interfaces.
- Automatic synthesis of the control logic to manage complex concurrency which is the most error prone part of RTL design.
- High level constructs for types, with very flexible type parameterization and strong static type-checking.

A.2 Bluespec System Verilog Constructs

A.2.1 Rules

BSV expresses synthesizable behavior with rules instead of synchronous always blocks. Rules are powerful concepts for achieving correct concurrency and eliminating race conditions. A primary feature of rules in BSV is that they are atomic; each enabled rule can be considered individually to understand how it maintains or transforms state. Atomicity allows the functional correctness of a design to be determined by looking at each of

the rules in isolation, without considering the actions of other rules. This one-rule-at-a-time semantics greatly simplifies the process of determining the functional correctness of a design.

In the hardware implementation compiled by the BSV compiler, multiple rules will execute concurrently. The compiler ensures the actual behavior is consistent with the logical behavior, thus preserving functional correctness while achieving performance goals.

Components of Rules

Rule Condition: A boolean expression which determines if the rule body is allowed to execute or not.

Rule Body: A set of actions which describe the state updates that occur when the rule fires.

Properties of Rules:

Execution of a rule w.r.t to all other rules is instantaneous, complete and ordered.

Instantaneous: All the actions in the rule body occur at a single, common instant and there is no sequencing of actions within a rule.

Complete: When the rule fires, the entire body of the rule executes. There is no concept of partial execution of a rule body.

Ordered: Each rule execution conceptually occurs either before or after every other rule execution, but never simultaneously.

A.2.2 Modules

A module consists of three things: state, rules that operate on that state, and an interface to the outside world (surrounding hierarchy). A module definition specifies a scheme that

can be instantiated multiple times.

A.2.3 Methods

Signals and buses are driven in and out of modules using methods. These methods are grouped together into interfaces. There are three kinds of methods:

Value Methods: Take 0 or more arguments and return a value.

Action Methods: Take 0 or more arguments and perform an action (side-effect) inside the module.

ActionValue Methods: Take 0 or more arguments, perform an action, and return a result.

A.2.4 Interfaces

Interfaces provide a means to group wires into bundles with specified uses, described by methods. An interface is a reminiscent of a struct, where each member is a method. Interfaces can also contain other interfaces.

A.3 Building a design in Bluespec System Verilog

The various steps involved in building a design in BSV is shown in Figure: A.1.

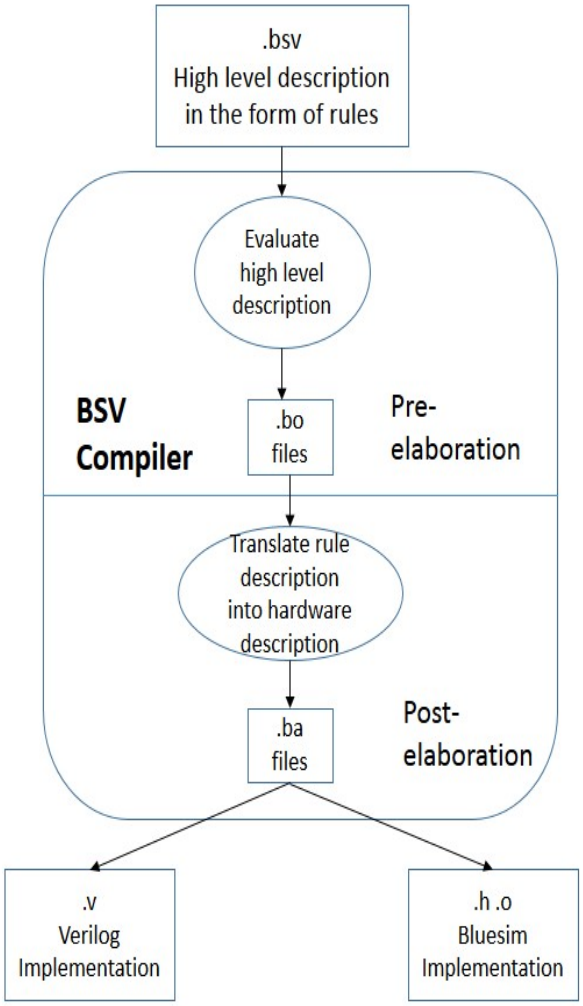


Figure A.1: Building a design in BSV

1. The designer writes the BSV code and it may contain Verilog, VHDL and C components.
2. The BSV code is compiled into a Verilog or a Bluesim specification. This step has 2 stages:
   - Pre-elaboration parsing and type checking.
   - Post-elaboration code generation.
3. The compiled output is either linked to a simulation environment or processed by synthesis tool.