openHMC

an open-source
Hybrid Memory Cube Controller

Computer Architecture Group, University of Heidelberg

in partnership with Micron Foundation

openHMC documentation Rev 1.3

Contents

1 » About openHMC

1.1 What is openHMC?

openHMC is an open-source project developed by the Computer Architecture Group (CAG) at the University of Heidelberg in Germany. It is a vendor-agnostic, AXI-4 compliant Hybrid Memory Cube (HMC) controller that can be parameterized to different data-widths, external lane-width requirements, and clock speeds depending on speed and area requirements. The main objective of developing the HMC controller is to lower the barrier for others to experiment with the HMC, without the risks of using commercial solutions. This project is licensed under the terms and conditions of version 3 of the Lesser General Purpose License[1].

1.2 About The Hybrid Memory Cube

The HMC is memory that is built of stacked DRAM, organized in independent sections, so called vaults. Figure 1.1 shows an abstract view of the structure of an HMC. It integrates all DRAM-related management circuits and therefore off-loads the user from any 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. More information on the HMC and its specification are available at the official Hybrid Memory Cube Consortium (HMCC) website www.hybridmemorycube.org.

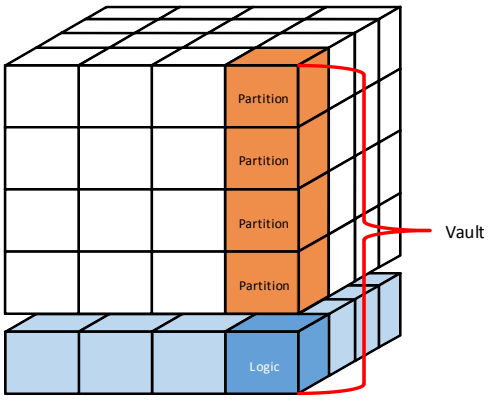

Figure 1.1: HMC: Abstract View

1.3 The openHMC Controller

The openHMC controller is presented as a high-level block diagram in Figure 1.2. The asynchronous input and output FIFOs allow the user to access the controller from a different clock domain. On the transceiver side, a registered output holds the data reordered on a lane-by-lane basis; allowing seamless integration with any transceiver types. A register-file provides access to control and monitor the operation of the controller.

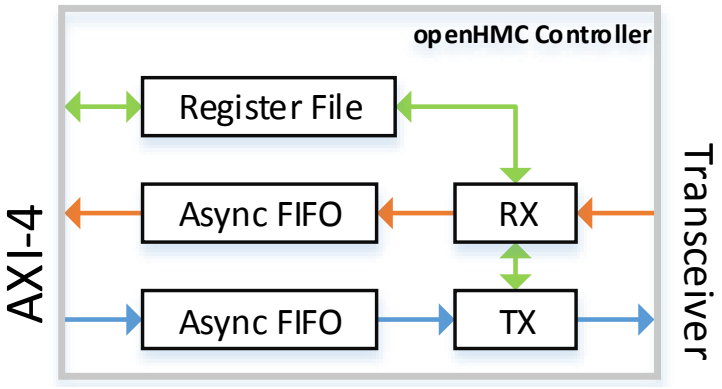

Figure 1.2: openHMC Memory Controller Block Diagram

1.4 Features

The openHMC memory controller implements the following features as described in the HMC specification Rev 1.1 [2]:

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

1.4.1 Supported Modes

Currently the following configurations are supported (8 or 16 lanes):

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

Other configurations may require specific CRC implementations and/or initialization schemes. For a more detailed overview of commonly used configuration modes see Chapter 4.

2 » Module Description

This chapter describes the Verilog modules of the openHMC package. The directory structure is attached in Appendix C. Note that the openHMC testbench is introduced separately in Chapter 6.

2.1 Top Module (hmc_controller_top.v)

The HMC controller top module instantiates and connects all logical sub-modules and does not contain any logic itself. It provides the AXI-4 , Transceiver and Register File interfaces. Figure 2.1 shows a more detailed view of the memory controller top level including the two clock domains and main interface signals. For a full interface specification refer to Chapter 3. The memory controller is often also referred to as 'Requester' and the data flow from host to HMC is called downstream traffic, or transmit direction (TX). The requester issues request packets and receives responses. On the other hand, the HMC is the 'Responder' and any traffic flowing in host direction is called upstream traffic, or receive direction (RX). The responder receives and processes requests, and returns responses if desired by the request type. In the following, all sub-modules are described in the order they are logically passed by a request/response transaction.

2.2 Asynchronous RX and TX FIFOs (hmc_async_fifo.v)

The asynchronous FIFOs connect the user logic in the clk_user clock domain to the HMC controller in the clk_hmc clock domain. Both FIFOs appear as an AXI-4 Stream Protocol Interface to the user. The full interface specification can be found in Chapter 3.

2.3 TX Link (tx_link.v)

The TX Link has two main interfaces, that is the input FIFO interface to receive HMC packets and the output register stage which provides scrambled and lane-by-lane re-ordered data FLITs to connect the transceivers. The user must generate HMC packets within the user logic, including the 64bit header. Also, the user is responsible for operational closure with TAGs, if desired. Note that an unsupported command or a dln/lng mismatch may produce undefined behavior in the current implementation. The 64bit tail must be set all to zero

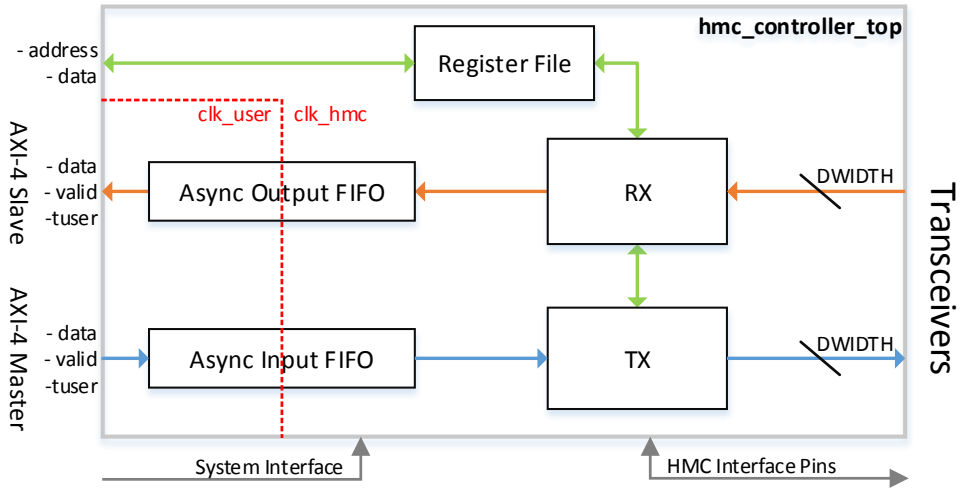

Figure 2.1: Detailed view of the Memory Controller Top Module

since it will be filled in the TX Link. Internally, the HMC controller uses register stages to encapsulate logically-independent units, and to avoid critical paths due to excessive use of combinational logic. The main control function is implemented as the following Finite State Machine (FSM):

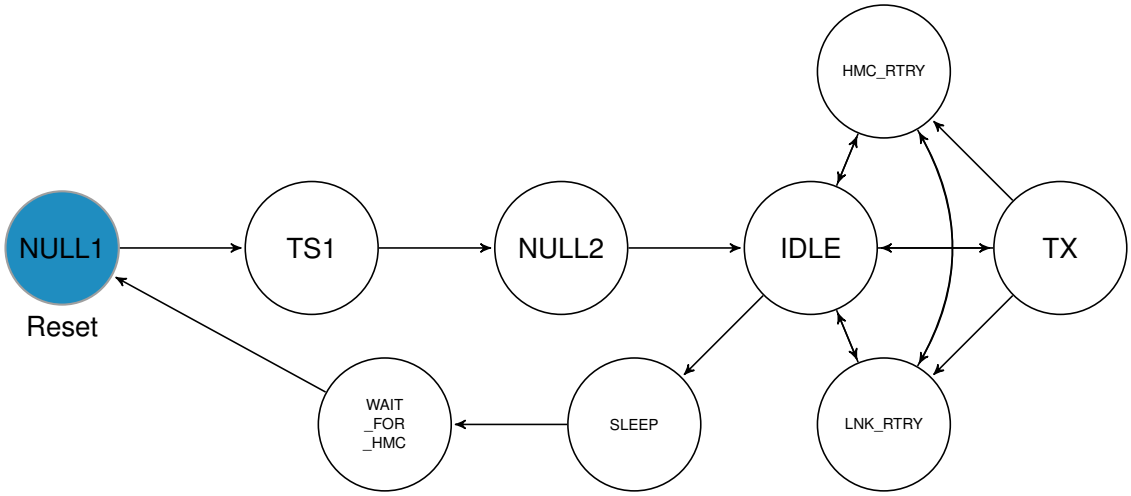

Figure 2.2: TX FSM

States and transitions are listed in Table 2.1 and Table 2.2. The next states are listed in the order of their priority. By default, the current state is maintained. For a better understanding of the initialization steps necessary after power-up refer to Section 4.2.

When in TX state, FLITs are processed as implied by the blue path in Figure 2.3. Register File (RF) signals and such that are driven by the RX link are represented by green colored, control signals by gray colored arrows. The operation of the TX link can be summarized as followes: 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

Table 2.1: TX FSM State Table

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

Table 2.2: TX FSM Transition Table

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

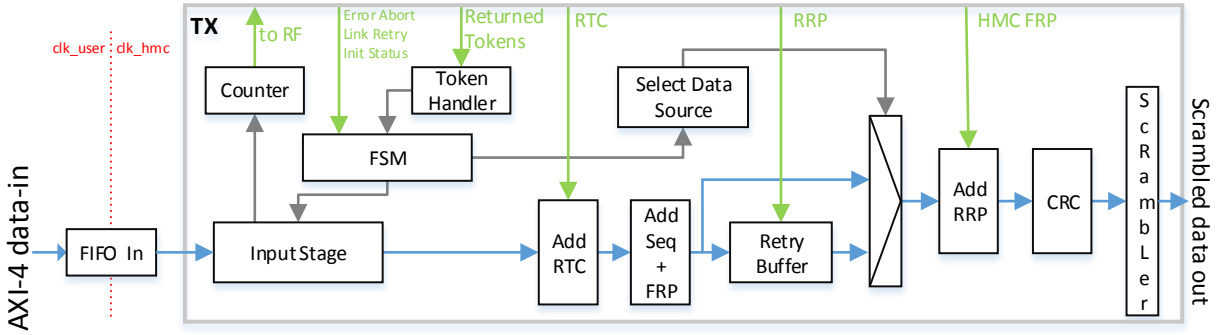

Figure 2.3: TX Link Diagram

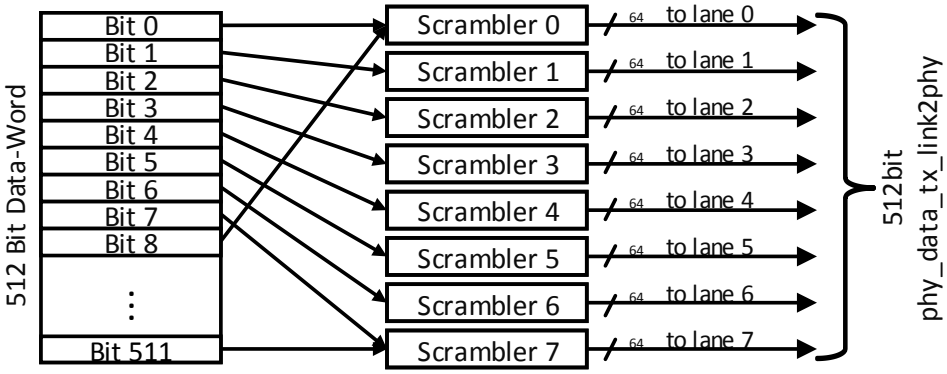

Figure 2.4: Data-Reordering: 4FLIT/512bit example

is decremented. When the token count is sufficient and no other interrupt occurs, the Return Token Count (RTC) is added to return tokens to the HMC, which indicates the number of FLITs that passed the RX input buffer. Afterwards, the Sequence Number (SEQ) and the Forward Retry Pointer (FRP), which is also the retry buffer read pointer, are added. At this point, all FLITs are also written to the retry buffer. If there is a link retry request (signaled by tx_link_retry_request) data is retransmitted out of the retry buffer instead of the regular datapath. Eventually the Return Retry Pointer (RRP) which is the last received HMC FRP is added, the CRC generated, and data is scrambled and reordered on a lane-by-lane basis depending on the configuration (NUM_LANES and DWIDTH). Figure 2.4 shows an example for a 512-bit / 8-lane configuration where each transceiver connects to 64bit of the parallel output stage.

2.3.1 TX Retry Buffer (hmc_ram.v)

The retry buffer holds a copy of each FLIT transmitted for possible retransmission. NULL FLITs and flow packets, except TRET, are not subject to flow control and retransmission, and are therefore not saved in the retry buffer. The retry buffer actually consists of FPW times 128-bit RAMs so that each FLIT can be addressed independently. One address, which

Table 2.3: RAM Configurations

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

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). Table 2.3 summarizes the RAM properties for different data-width configurations. Note that a 6-FLIT configuration results in reduced RAM capacity since 6 is not a power of 2 and therefore the next higher of LOG_FPW must be chosen, leaving some addresses unused. The least significant bits address the target RAM while the remaining bits refer to a specific FLIT within that RAM. The entire value is called FRP, and at the same time is the RAM write pointer. As a result of this addressing scheme, FRPs are not generated consecutively but still incremental, as packets may consist of more than one FLIT. The read pointer of the RAM moves with each RRPs received at the RX Link, following the write pointer and therefore excluding potential FLITs from retransmission. The link retry mechanism is described in Section 4.5.

2.3.2 Scrambler (tx_scrambler.v)

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

2.3.3 Lane Run Length Limiter (tx_run_length_limiter.v)

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

2.3.4 CRC (tx_crc_combine.v)

The CRC architecture was specifically chosen to scale with different data-widths. As can be seen in Figure 2.5 it consists of one 128-bit CRC per FLIT (crc_128_init). While the CRCs

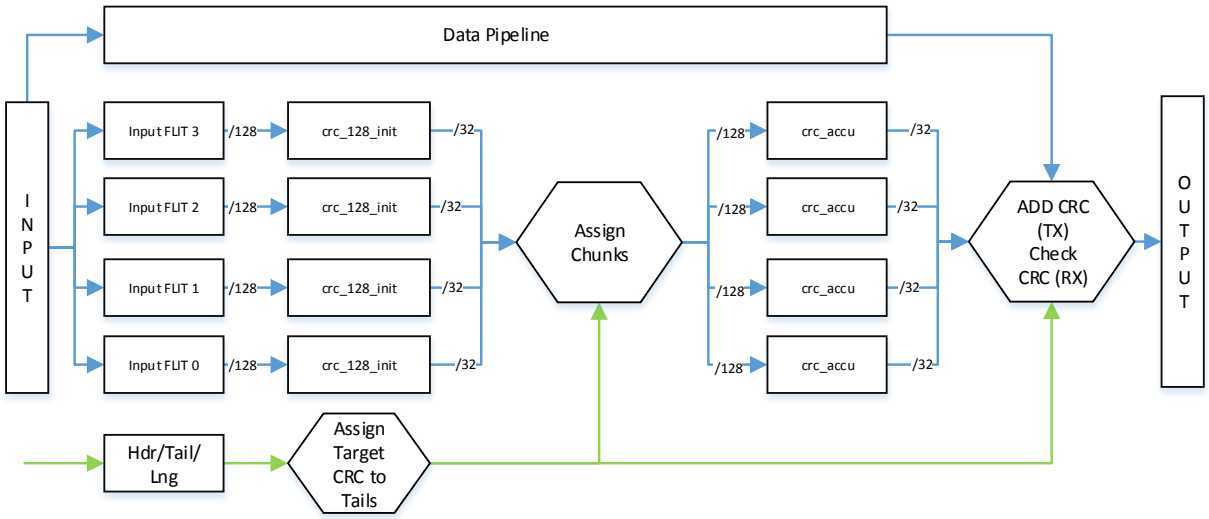

Figure 2.5: Scalable CRC Architecture: FPW=4 Example

are calculated a specific logic assigns the targeted CRC to the tail of the corresponding packet. After the CRCs are calculated all 32-bit remainder that belong to the same packet are shifted to a dedicated accumulation CRC stage (crc_accu), where the remainders form the actual CRC within a single cycle. Finally, the output CRCs are added to the tail of the packets.

2.3.5 General Notes on TX Link

The TX link only returns one flow packet per cycle, which is sufficient and an easy way to save some logic. However, (re-)initialization for instance will take some additional cycles to transmit all available tokens since only 31 tokens may be returned within a single Token Return (TRET) packet.

2.4 RX Link (rx_link.v)

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

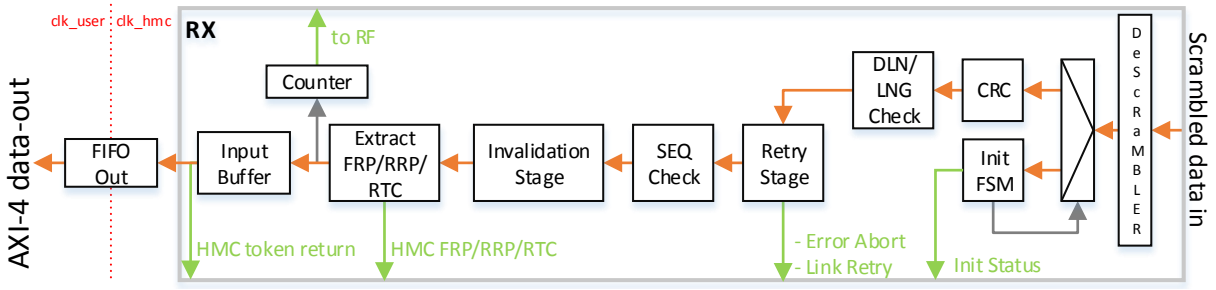

Figure 2.6: RX Link Diagram

2.4.1 CRC (rx_crc_compare.v)

The rx_crc_compare module is very similar to the tx_crc_combine instantiated in the TX Link. The biggest difference is that the CRCs are not added to the tail of a packet at the end of the data pipeline, but compared. The corresponding poisoned or error flag for the tail of the faulty packet is set if a mismatch occurs. Additionally, the data pipeline of this module holds information bits for valid/header/tail FLITs as this information will be used in the RX link.

2.4.2 Descrambler (rx_descrambler.v)

The rx_descrambler module is instantiated once per lane and is self-seeding, which means that it automatically determines the correct value for the internal LFSR. As the seed for a descrambler is determined, the descrambler is 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.

2.4.3 Input Buffer (sync_fifo_simple.v)

The input buffer holds 2**LOG_MAX_RTC entries, where each entry is as wide as the datapath (DWIDTH). This results in more resource utilization, but allows a series of 2**LOG_MAX_RTC cycles, carrying one valid FLIT each to be shifted-in without a need for additional buffer distribution and utilization logic. Each valid FLIT at the buffer output returns 1 token to the TX link on a shift_out event. These tokens will be returned as RTC to the HMC.

2.5 Register File (hmc_controller_8x_rf.v and hmc_controller_16x_rf.v)

The Register File features three main types of registers: Control, Status, and Counter. Control registers directly affect the memory controller or HMC operation. Status registers can be used to monitor the status of the memory controller, especially during initialization. Counters allow performance measurement. For a full list of available registers, see Appendix B. Note that there are several 'reserved' fields which are not listed in the table of registers. These reserved fields provide some space to add additional information, and also align the fields within a register. These unused fields will be tied to constant 0 during synthesis. There are two different RFs that provide the same registers, but a few different signal widths depending on the HMC link configuration (half-width/full-width). The correct RF is instantiated automatically according to the NUM_LANES parameter.

2.6 Header Files

The following header files are present:

hmc_field_functions.h

hmc_field_functions contains useful functions that return fields such as length or the CRC out of HMC headers or tails

3 » Interface Description

This chapter contains an interface description for the top module hmc_controller_top.v. Since the memory controller can be controlled through parameters, most of the signal-widths depend on the configuration used. The hmc_controller_top module contains a set of parameters that can be used to override the default configuration. All parameters used are listed in Table 3.1.

Table 3.1: Configuration Parameters

Parameter	Description	Default
FPW	Desired data-width in FLITs (1FLIT = 128bit). Valid: 4/6/8	4
LOG_FPW	Log of the desired data-width in FLITs	2
DWIDTH	FPW*128, width of the databus in bits	512
LOG_NUM_LANES	Log of the amount of HMC Lanes. Valid: 3/4	3
NUM_LANES	Amount of HMC Lanes (8 or 16)	8
NUM_DATA_BYTES	FPW*16, defines the AXI-4 TUSER bus width in bytes	64
HMC_RF_WWIDTH	Register file rf_write_data bus size in bits	64
HMC_RF_RWIDTH	Register file rf_read_data bus size in bits	64
HMC_RF_AWIDTH	Register file rf_address bus size in bits	4
LOG_MAX_RTC	Log of the max RX input buffer space in FLITs	8
HMC_RX_AC_COUPLED	Set to 0 if Controller TX is DC coupled to HMC RX	1

3.1 System Interface

The controller top module (hmc_controller_top) expects a clock and a reset per clock domain, where each reset must be synchronous to the corresponding clock. Most likely, clk_hmc and the parallel transceiver clock domain will be sourced by the same driver. The user clock clk_user may be any clock faster than clk_hmc. Figure 3.1 shows the system interface. Note that both resets are active low.

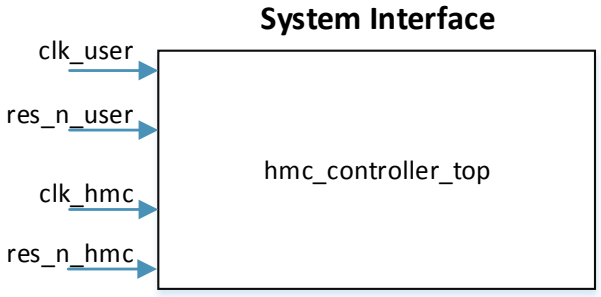

Figure 3.1: System Interface Diagram

3.2 HMC Interface

The HMC provides the four signals presented in Figure 3.2. Note that the HMC reset P_RST_N and the both power-reduction pins LxRXPS and LxTXPS are active low. The active low fatal error indicator FERR_N is not connected it this revision of the memory controller and is considered 'don't care'.

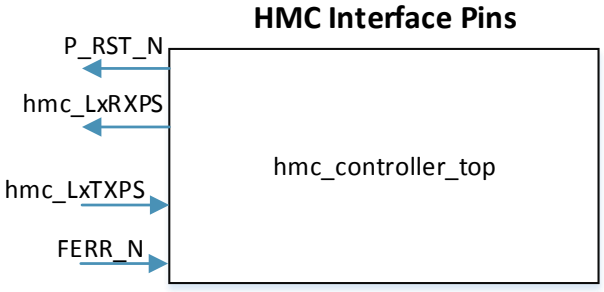

Figure 3.2: HMC Interface Pins Diagram

3.3 AXI-4 Stream Protocol Interface

Both AXI-4 interfaces comply with the ARM AMBA AXI-4 Interface Protocol Specification v1.0 [3]. However, not all signals are used. Figure 3.3 provides an interface diagram of the master and slave interfaces used in this implementation. The use and the corresponding size of these signals is described below.

TREADY 1 bit

- TX: Memory controller is ready to sample TDATA and TUSER
- RX: Valid data on TDATA and TUSER

TVALID 1 bit

- TX: TDATA and TUSER are sampled on TX when TVALID=1 and TREADY=1. TVALID may be held high even when TREADY=0.

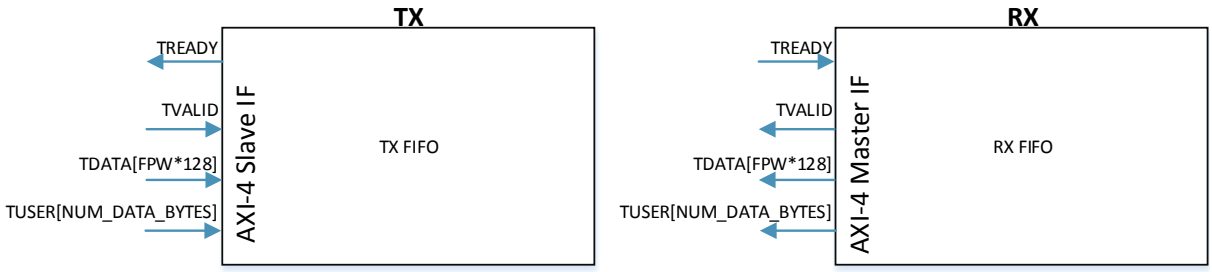

Figure 3.3: AXI-4 Interface Diagram

- RX: TDATA and TUSER are valid when TVALID=1. TREADY may be held high even when TVALID=0. TDATA and TUSER will not change when TREADY=0.

TDATA FPW*128 bit

The TDATA bus expects complete HMC request packets, starting with the 64bit header followed by data FLITs. The user is responsible to populate all request header fields (see Figure 3.4 or refer to the HMC documentation, chapter 'Request Commands'). Note that the TAG field is optional, but required for operational request/response closure. The tail must be set to all zeroes. Figure 3.5 shows an example transaction of multiple different packet types. Packets may start at any 128-bit/ FLIT border. 'Bubbles' between packets are allowed as long as the corresponding valid bit(s) is/are kept low. All FLITs of a packet must be transmitted throughout consecutive FLITs. Also when a packet spreads over multiple 512-bit cycles, TVALID must be held high until the entire packet (including its tail) was transmitted. On RX, the memory controller outputs complete HMC response packets. Data is valid when TVALID=1 and the output will not change while TREADY=0. Contrary to TX, the user has full control on when TREADY is set. When seen a response header the packet does not need to be sampled consecutively throughout its tail.

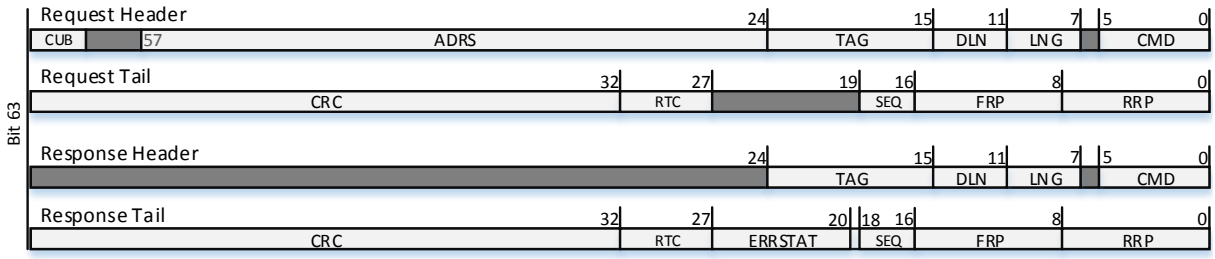

Figure 3.4: HMC Header and Tail

TUSER NUM_DATA_BYTES bit

The user is responsible to set the following information on the TX TUSER bus respec-

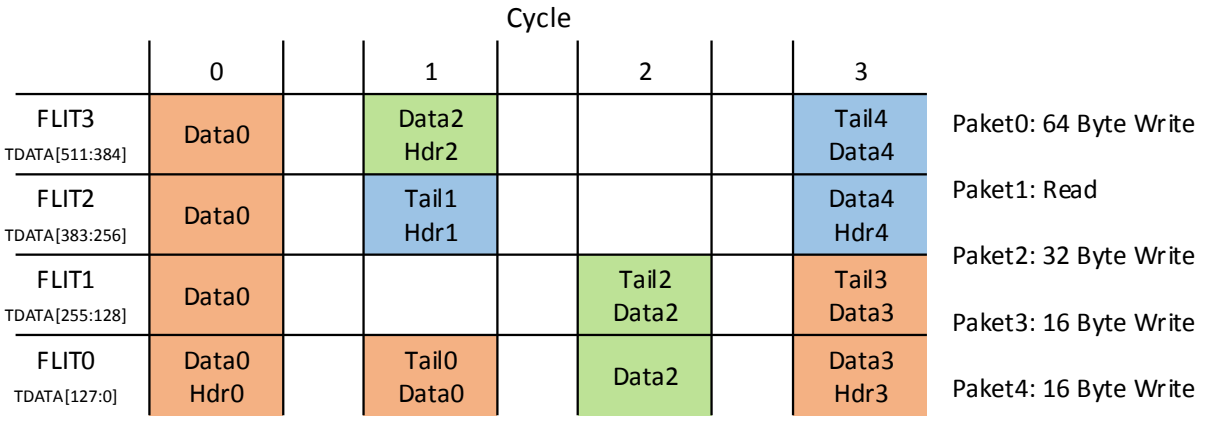

Figure 3.5: Example transactions on the AXI TX TDATA bus for FPW=4

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

Figure 3.6: TUSER Example for FPW=4

tively the controller provides these information at the RX TUSER bus:

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

Every FLIT on the TDATA bus corresponds to one bit in the valid, hdr, and tail fields on TUSER. FLIT 0 at TDATA[127:0] is defined by valid[0] (TUSER[0]), hdr[0](TUSER[FPW]), and tail[0](TUSER[2*FPW]).

Example:

TDATA holds a header on FLIT position 0 (TDATA[127:0]). Set hdr[0] respectively TUSER[FPW] to 1. Since a header is a valid FLIT, set valid[0] / TUSER[0] to 1. This scheme applies to all FLITs on the TDATA bus. Figure 3.6 illustrates how to set the TUSER signal according to the content of the TDATA bus in Figure 3.5.

Important

To guarantee proper interface operation, all FLITs of a packet must be shifted in continuously on TX, this means without any 'bubble' FLITs or cycles in between. There is no constraint on 'bubbles'/NULL FLITs/NULL cycles between packets. Additionally the frequency of the user clock driving the AXI-4 interface must be equal to or higher than clk_hmc.

3.4 Transceiver Interface

The TX Link provides a DWIDTH wide register output phy_data_tx_link2phy with scrambled and lane-by-lane ordered data, driven by clk_hmc. Hence the bits [(1*LANE_WIDTH)-1:(0*LANE_WIDTH)] contain data for lane 0, [(2*LANE_WIDTH)-1:(1*LANE_WIDTH)] data for lane 1 and so on. An additional input phy_ready should be connected to transceivers 'reset_done' (or similar) to allow monitoring of the transceiver status. The RX Link's data input register phy_data_rx_phy2link expects input data by the receivers using the same ordering as explained for the TX Link. Lane reversal is detected and applied in the RX Link and does not affect ordering. Additionally, the RX Link outputs bit_slip wires, one per lane used to compensate lane-to-lane skew during initialization. Connect these to the corresponding transceiver. All signals are summarized in Table 3.2. Listing 3.1 shows how to connect the transceiver lanes in a DWIDTH=512bit and NUM_LANES=8 configuration, with a lane-width of 512bit/8lanes=64 bits per lane.

Table 3.2: Transceiver Interface Signals

Signal	Width	Description
phy_data_tx_link2phy	DWIDTH	Lane by lane ordered output
phy_data_rx_phy2link	DWIDTH	Lane by lane ordered input
phy_ready	1	Signalize that the transceivers are ready
bit_slip	HMC_ NUM_ LANES	Bit_slip is used to compensate lane to lane skew. Bit_slip is controlled by the RX Block for each lane individually

Listing 3.1: Transceiver Connectivity Example

```
wire [DWIDTH-1:0]     tx_data;
wire [DWIDTH-1:0]     rx_data;
wire [NUM_LANES-1:0]  rx_bit_slip;

hmc_controller_top #(...) openhmc_I (
```

```
        ⋮
        .phy_data_tx_link2phy(tx_data),
        .phy_data_rx_phy2link(rx_data),
        .phy_bit_slip(rx_bit_slip),
        ⋮
        );

transceiver_top #(...) transceiver_l (
        ⋮
        .lane0_tx_data(tx_data[63:0]),
        .lane1_tx_data(tx_data[127:64]),
        ⋮
        .lane0_rx_data(rx_data[63:0]),
        .lane1_rx_data(rx_data[127:64]),
        ⋮
        .lane0_bit_slip(rx_bit_slip[0]),
        .lane1_bit_slip(rx_bit_slip[1]),
        ⋮
        );
```

3.5 Register File Interface

A Register File module allows to control and monitor the operation of the memory controller. The interface signals are shown in Figure 3.7 and described in Table 3.3. First the target address must be applied. For a write, write_data must hold the 64-bit value to be written. Data is sampled when write_enable is asserted. For a read the read_enable signal must be asserted instead. Each operation is confirmed by the access_complete signal set for

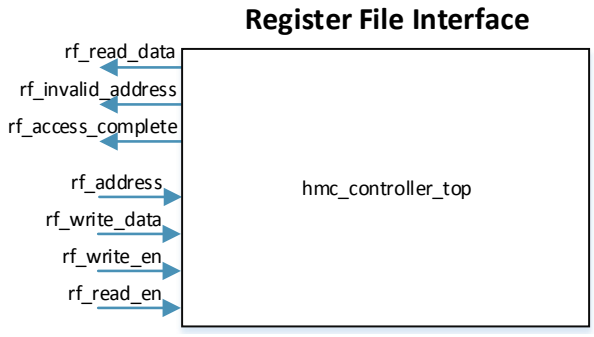

Figure 3.7: Register File Interface Diagram

Table 3.3: Register File Interface Signals

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

Table 3.4: Register File Address Map

Register	Address	Description
status_general	0x0	General HMC Controller Status
status_init	0x1	Debug register for initialization
control	0x2	Control register
sent_p	0x3	Number of posted requests issued
sent_np	0x4	Number of non-posted requests issued
sent_r	0x5	Number of read requests issued
poisoned_packets	0x6	Number of poisoned packets received
rcvd_rsp	0x7	Number of responses received
counter_reset	0x8	Reset all counter
tx_link_retries	0x9	Number of Link retries performed on TX
errors_on_rx	0xA	Number of errors seen on RX
run_length_bit_flip	0xB	Number of bit flips performed due to run length limitation

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 3.8 provides an example for a register write followed by a read to address 0x10. Refer to Table 3.4 for the address mapping. For a full listing of all fields within the RF see Appendix B.

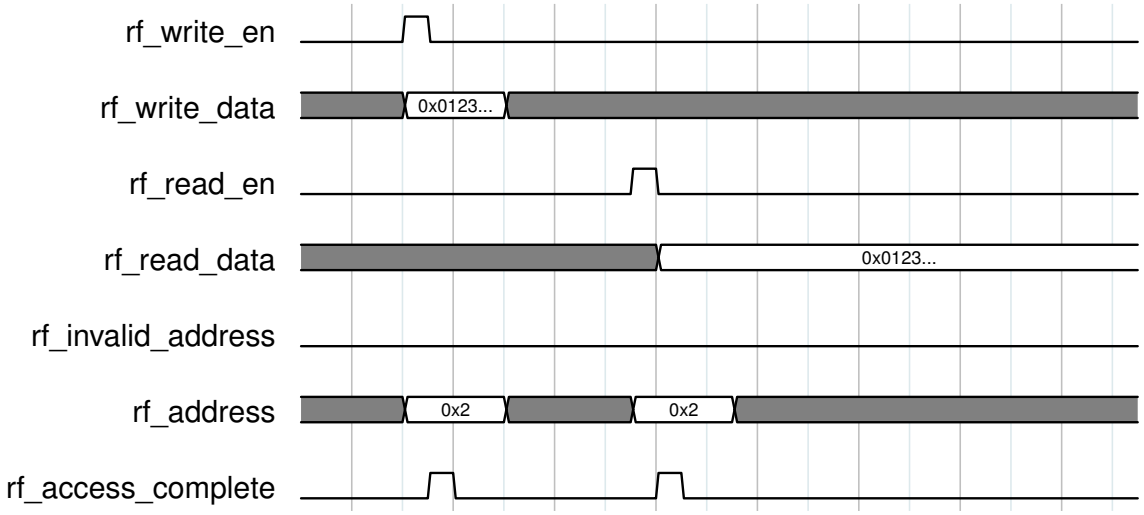

Figure 3.8: Register File Access: Write and read register 0x2

4 » Configuration and Usage

The following chapter provides information on how to properly configure and use the openHMC controller.

4.1 Clocking and Reset

Always keep both reset signals, res_n_user and res_n_hmc synchronous to their corresponding clock. Although the ‘ifdef ASYNC_RES macro is used for all clock-triggered always@ blocks, asynchronous reset should not be used where the target registers do not provide a dedicated asynchronous reset path. This is the case for FPGAs.

4.2 Power Up and (Re-)Initialization

As soon clk_hmc is stable and the low-active res_n_hmc has been de-asserted, initialization can begin. The p_rst_n bit in the control register is used to drive the active low HMC reset signal P_RST_N. The general HMC initialization process is shown in Figure 4.1. In this example I2C is used to load the internal HMC registers during the register load period (JTAG may be used instead, refer to the HMC documentation [2]). Note that HMC register load is not performed by the openHMC controller. As soon as the configuration is done and the ’init continue bit’ in the HMC internal registers is set, the user must also set the hmc_init_cont_set bit in the control register to allow the descramblers to lock. Any delay in doing so may also delay the initialization process. No other user activity is required until the link_is_up flag in

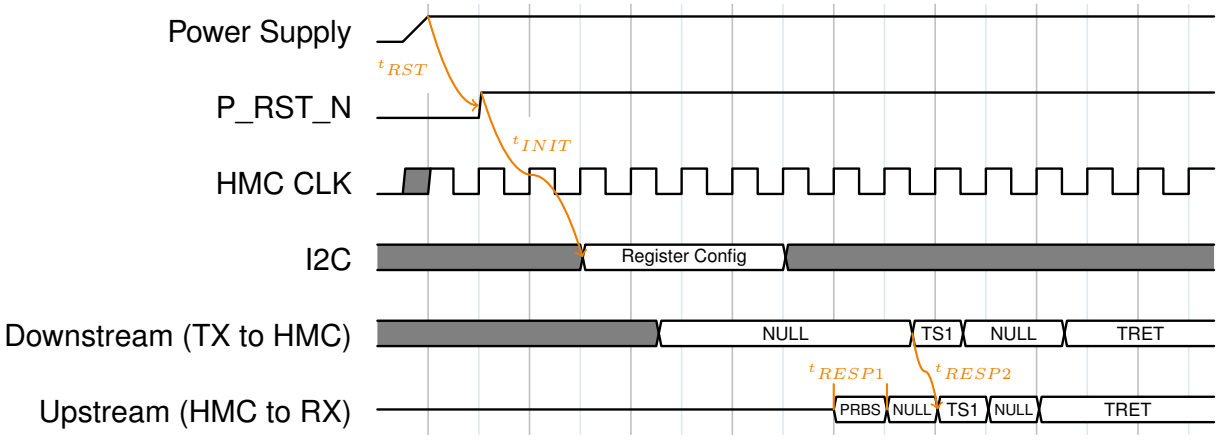

Figure 4.1: TX-Link: Initialization Timing

the RF is set. The AXI-4 user interface may remain in reset during the initialization process. Figure 4.2 provides the essential steps for the controller power up. Optionally the user can set the values provided in Table 4.1 prior the de-assertion of res_n_hmc which directly affect the initialization process.

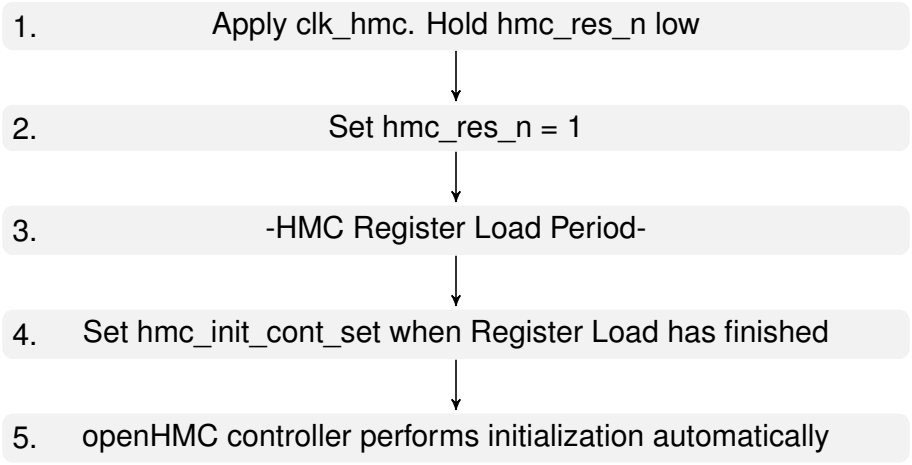

Figure 4.2: openHMC Controller Power Up Steps

Table 4.1: Configuration Parameters

Register	Valid values	Description
RX_tokens_av	$0 \leq 1023$	Set the available token space in the RX input buffer. Note: LOG_MAX_RTC must be adjusted so that 2**LOG_MAX_RTC is greater or equal to RX_tokens_av
bit_slip_time	$0 \leq 255$	Cycles between two bit-slips (Refer to the target transceiver user guide)
scrambler_disable	0/1	Disable scrambler and descrambler (can be useful for testing/debugging)

AXI4 Interface

The AXI4 user interface is considered 'don't care' as long as res_n_user is held low. No action is this interface is required for power up and initialization. However, it may be activated at any time.

4.3 Sleep Mode

Sleep mode can be safely entered when all in-flight transactions are complete and the TX Block is in IDLE state. For instance, the performance counter in the RF can be used to track the status of outstanding requests. To request sleep mode, the corresponding

set_hmc_sleep field in the RF control register must be set. The HMC will acknowledge sleep mode by setting the hmc_LxTXPS pin low. To exit sleep mode, de-assert set_hmc_sleep. The sleep_mode field within the RF status_general register may be used to monitor the entire process. Upon completion, the link is re-initialized as shown in Figure 4.1, except the need to exchange initial TRETs as memory contents within the HMC are maintained during sleep mode.

4.4 Link retraining

When detecting an unacceptable rate of link error monitored by the link_retries counter, sleep mode should be entered and exited to retrain the link. All steps described in Section 4.3 apply.

4.5 Link Retry

As soon as a link error occurs, the respective receiver of the faulty packet enters the 'Error Abort Mode'. There are two types of link retries that are described in the following. For a better understanding, Figure 4.3 illustrates the flow of pointer between the memory controller and the HMC. Note that both endpoints, memory controller and HMC, generate and check FRP's and RRP's the same way.

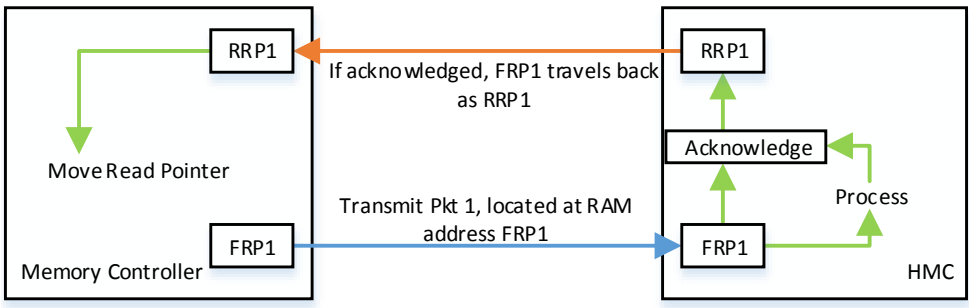

Figure 4.3: Pointer Flow

TX Link Retry

In case of an error on the TX path from requester to responder, the HMC will request a link retry. Subsequent received packets arriving at the HMC are dropped, and no header/tail values are extracted. The HMC then 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). 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). 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 where re-transmitted. Upon completion the RAM read address returns to the last received RRP. Re-transmitted packets may therefore be re-transmitted again if another error occurs. Figure 4.4 shows the TX link retry mechanism.

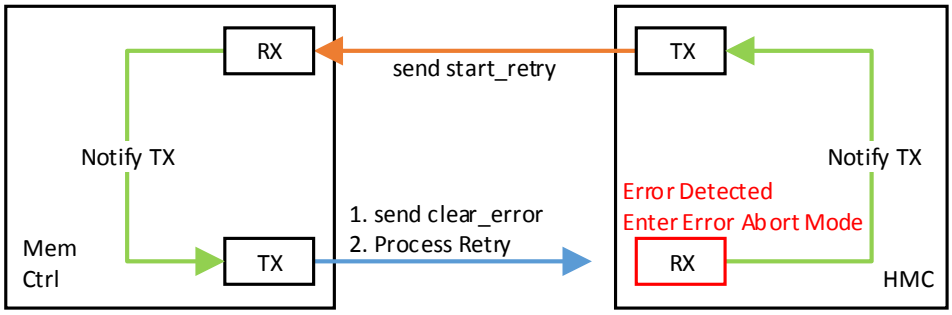

Figure 4.4: TX Link Retry

HMC Retry

In case of an error on the RX path from responder to requester, the RX link will request a link retry. The TX link will than send start_retry packets whereupon the responder will start to re-transmit all packets that were not acknowledged by the RRP yet. Meanwhile, the RX link remains in the so called error_abort_mode where all subsequently incoming packets are dropped. 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 4.5 shows the TX link retry mechanism.

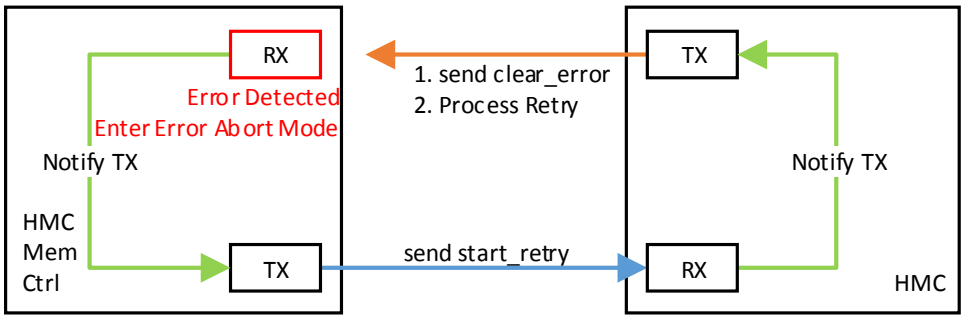

Figure 4.5: HMC Retry

Link Retry

For correct link retry operation, equal to or more irtry packets (both types) must be issued than the respective receiver expects. This requirement applies to both, requester and responder. The corresponding irtry_to_send value must be equal to or higher than irtry_received_threshold in the register file (default). The internal registers in the HMC must be set accordingly.

4.6 openHMC Configuration

According the configuration of the data-width (DWIDTH), half-width or full-width (NUM_LANES) and their respective lane speed, the configurations in Table 4.2 can be applied. Table 4.3 lists all valid parameter sets. The resulting clocking frequencies are calculated with:

$$\text{Frequency}[\text{MHz}] = \frac{\text{NUM\_LANES} * \text{LANE\_SPEED}}{\text{DWIDTH}}$$

Table 4.2: Possible Configurations

DWIDTH [bit]	NUM_LANES	lane speed [Gbits]	clk_hmc [MHz]
256	8	10	312.5
512	8	10	156.25
512	8	12.5	195.3125
512	8	15	234.375
512	16	10	312.5
768	8	15	156.25
768	16	15	312.5
1024	16	10	156.25
1024	16	12.5	195.3125
1024	16	15	234.375

Table 4.3: List of valid parameter sets

Desired DWIDTH [bit]	LOG_FPW	FPW
256	1	2
512	2	4
768	3	6
1024	3	8

Clocking

Assure that clk_user is equal to or higher than clk_hmc for proper AXI-4 interface operation

4.7 HMC Configuration

Input Buffer Token Count

By default the input buffer token count of the rx_link input buffer is set to 100'd. It can be changed using the rx_token_count register in the Register File control register, if desired. According to the maximum packet length of 9 FLITs, it must be set to 9 or more.

Maximum Packet Size

The user must not send any packets bigger than 'maximum block size' in the HMC Address Configuration Register is set to.

Token Count

To avoid misbehavior for any of the listed configurations, set the token count within the HMC token register to 33 or more

5 » Implementation

This section gives advice on key elements to consider in order to successfully implement the openHMC controller. It further presents example configurations that were already implemented and verified in an FPGA.

5.1 Design with the Core

As always, a good design practice is inevitable in order to successfully implement a design and close timing. Implementing the openHMC controller in a 2-FLIT/10Gbit configuration is not extremely challenging. However, when it comes to 1024bit datapaths and lane-speeds of 15Gbit/s, logical paths may fail for several reasons:

High fanout nets Candidates for very high fanout nets are global clocks or resets for example. Use clock or reset buffer at the input of hmc_controller_top or limit the loads by replicating heavy-loaded nets. Alternatively, reset conditions may be removed where applicable. This is especially the case for pipelined datapaths and registers that should hold logical zeroes at power-up. Refer to Section 5.3 for more information.

Non- or false constrainted clock-domain crossings Clock domain transitions, such as in asynchronous FIFOs, must be explicitly defined as asynchronous paths. This prevents the implementation tool from investigating the timing on these paths.

Routing congestion and overlapping nets Components with a high logic density such as the crc modules may be difficult to route, especially in a 1024bit/FPW=8 configuration. Solutions may be location constraints, additional pipelining, or the use of special implementation strategies.

5.2 Implementation Results

The openHMC controller was verified in simulation using multiple verification environments, including the Micron HMC Bus Functional Model (BFM). Additionally, it was successfully implemented and tested with the device(s) listed in Table 5.1. The Xilinx Vivado Design Suite 2014.3.1 was used as implementation tool, with the Vivado Default Synthesis and Implementations settings. LOG_MAX_RTC was set to 8. The run_length_limiter was used with its granularity set to 4.

Table 5.1: FPGA-Verified Configurations

ID	FPW	NUM _LANES	LANE _SPEED [Gbit]	clk_hmc [MHz]	Target
1	4	8	10	156.25	Xilinx Virtex Ultrascale XCVU095-FFVD1924-2-e-es1
2	4	8	12.5	195.3125	Xilinx Virtex Ultrascale XCVU095-FFVD1924-2-e-es1

In addition to the FPGA-verified configurations, the openHMC controllers was successfully implemented in an 8FLIT, 1024bit datapath, 16 lane, 12.5Gbit configuration.

5.2.1 Resource Utilization

Table 5.2 gives an overview over the approximate resource utilization for each implementation run listed in Table 5.1, matched by the ID. Note that the presented values are the results for the openHMC controller implemented in a larger design along with other components. Also, other implementation strategies may be used to target different area or performance goals.

Table 5.2: Resource Utilization

ID	LUTs combined [% device]	Registers [% device]	BRAM B36/B18 [% device]
1,2	16392 [3.04%]	13051[1.21%]	16 [0.92%]

5.3 Optimization Techniques

The following design advice can be used to reduce resource utilization, if applicable.

Disable Run Length Limiter

If HMC RX is DC coupled to Memory Controller TX, set the parameter HMC_RX_AC_COUPLED in hmc_controller_top to 0 in order to allow the synthesis tool to remove the run length limiter. DC coupled links are not subject to run length limitation.

Use of FPGA specific characteristics

Several optimizations may be applied depending on the target device. Some examples are:

Remove reset values If the target device is an FPGA that defaults register values to a logic zero at the end of configuration, some reset values may be removed. There is no

need to initialize these registers. Further constant propagation stages such as parts of the datapath in tx_link or rx_link may be connected without any need for a reset condition, hence reducing the fanout of the reset net, and decreasing routing effort and complexity. However, the need for a reset signal should be verified in functional simulation.

Use target specific components FPGA design tools such as the Xilinx Vivado Design Suite provide the possibility to generate optimized components such as FIFOs and RAMs tailored for the target device.

6 » openHMC Test Environment

With the release of openHMC revision 1.3 the package includes a Universal Verification Methodology (UVM) based test environment to demonstrate the functionality of the openHMC controller. It is designed and tested for the Cadence Incisive tool chain (NC Sim) version 13.10 and newer. Other simulators might be supported in the future.

6.1 Preparation

A few steps must be performed until the test environment is ready to use. Please follow the following instructions carefully and review the steps when experiencing problems. Refer to Appendix C for an overview of the directory/file structure.

1. Export the OPENHMC_PATH and OPENHMC_SIM environment variables. Example:
   export $OPENHMC_PATH=home/user/openhmc
   export $OPENHMC_SIM=home/user/openhmc/sim
   Alternatively source the script 'export.sh'.
2. Extract the BFM package
3. Copy the contents of the package to '$OPENHMC_SIM/bfm/'. The content of this folder should now contain the folders 'src', 'doc', and so on.
4. Open 'hmc_bfm.f' and change the all paths from src/ to $OPENHMC_SIM/bfm/src.

6.2 Run a Test

Navigate to $OPENHMC_PATH/sim/axi4/run and execute the runscript run.sh by typing './run.sh' for example. Table 6.1 lists all available arguments. A test in an 8FLIT and 16lanes configuration with high verbosity may be started with:

./run.sh -f 8 -l 16 -v UVM_HIGH

However, it is also possible to run the script without any arguments. In this case the design is automatically defaulted to an FPW=4 (512bit), NUM_LANES=8 configuration and 100 random packets will be sent. Besides the runscript the folder also contains a cleanup script 'clean_up.sh' which can be run to remove build files from previous simulation runs.

Table 6.1: Runscript Arguments

Argument	Requires Value	Description
-c		Clean up old build files
-d	X	Define a different target (advanced)
-f	X	FPW. Set the datapath width
-g		Start Simvision
-l	X	NUM_LANES. Set the number of lanes
-p	X	Defines the number of packets to be sent (default=100)
-s	X	Start the test with a different seed
-t	X	Specify a test (advanced)
-v	X	Verbosity of the debug output. Available values are UVM_NONE, UVM_LOW (default), UVM_MEDIUM, and UVM_HIGH
-?		Print usage help

Performance Considerations

The performance of the machine that runs the testbench may suffer from an excessive number of packets to be sent, defined by the -p argument of the runscript. It is recommended to restrict the number to less than 10000. However, further increasing the performance and capabilities of the test environment is of high priority for the openHMC team.

6.3 Testbench Configuration

While the basic openHMC controller configuration can be modified with the runscript arguments, more advanced configuration is available in the config.h header, located at $OPENHMC_PATH/sim/axi4/tb/src/targets/. Table 6.2 presents the available configuration set. When changing any of these value keep their limits in mind. The token counts for example must not exceed a value of 1023.

Table 6.2: Configuration Set

Name	Description	Limits
RX_TOKENS	Set the RX Link input buffer space in FLITs	$25 \leq x \leq 1023$
HMC_TOKENS	Set the BFM input buffer space in FLITs	$25 \leq x \leq 1023$
HMC_CUBID	Set the HMC Cube ID	$0 \leq x \leq 7$

6.3.1 Lane Polarity/Reversal and Routing Delays

The testbench allows to randomly insert delays and polarity inversion on single lanes, as well lane reversal per link. All these options are turned off by default. However, open 'dut_openhmc_behavioral.sv' and change the value(s) listed in Listing 6.1 as desired (0=disabled/false ,1=enabled/true).

Listing 6.1: Enable Lane Delays/Polarity/Reversal

```
bit disable_lane_delays     = 1;
bit disable_lane_polarity   = 1;
bit disable_scramblers      = 0;

localparam LANE_REVERSAL_TX = 0;
localparam LANE_REVERSAL_RX = 0;
```

6.4 Test Environment

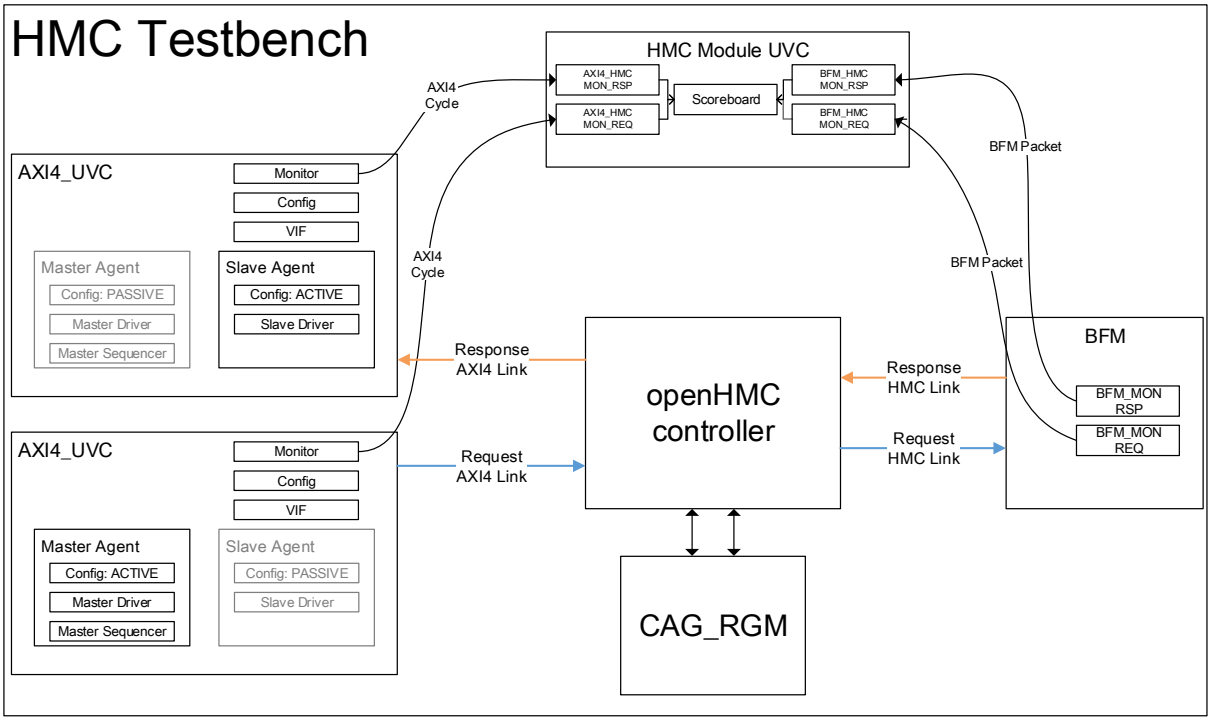

Figure 6.1: HMC Testbench

The UVM based test environment is presented in Figure 6.1. It consists of the following components:

AXI4 UVC Used to verify the AXI4 interface. Depending on the purpose the AXI4 UVC

creates a master agent that generates packets and drives AXI4 cycles into the Device under Test (DUT) respectively a slave agent that receives packets.

Module UVC The module UVC contains the Scoreboard, which contains 2 sets of input-analysis channels. One set is used to check packets on the AXI4 interface. The second set collects and checks packets at the openHMC HMC interface. Each set contains a request-, and a response analysis input. Packet types are defined in the base type package (hmc_packet.sv). The scoreboard checks all posted and non-posted data packets, TAGs included.

BFM The Micron BFM

CAG_RGM UVC Simulates the Register File access

All these components are instantiated within the HMC testbench (hmc_tb.sv)

6.4.1 Simple Test

The provided 'simple test' executes AXI4 master sequences, where one master sequence corresponds to a single randomized HMC packet on the openHMC AXI4 TX interface. By default 100 random packets will be transmitted. A different value can be provided with the '-p' argument when executing the runscript. Furthermore the maximum number of packets per AXI4 cycle can be constrainted by the variable 'max_pkts_per_cycle' in 'simple_test_seq.sv' as shown in Listing 6.2. By default it is set to FPW, meaning that a maximum of FPW packets may occur within a single cycle, e.g. a 16Byte Write packet and 2 Read packets within a single cycle for FPW=4.
More tests will follow in upcoming releases, including random error generation, sleep-mode demonstration, and others.

Listing 6.2: Change the 'sequence' definition

```
`uvm_create_on(requests, p_sequencer.axi4_req_seqr)
requests.num_packets = `NUM_PACKETS; //defined with -p
requests.max_pkts_per_cycle = `FPW; //Constrain me
`uvm_rand_send(requests)
```

6.4.2 Test Procedure

The simple test itself can be divided into three phases as shown in Figure 6.2. After the test is started the openHMC controller is configured in the hmc_init_seq initialization sequence. As soon as the link is up (signalized by the link_up bit in the register file 'status register'), the actual test is started. First, random packets will be generated. Non-posted request will be assigned a TAG. These requests are queued, with random 'bubble' FLITs between

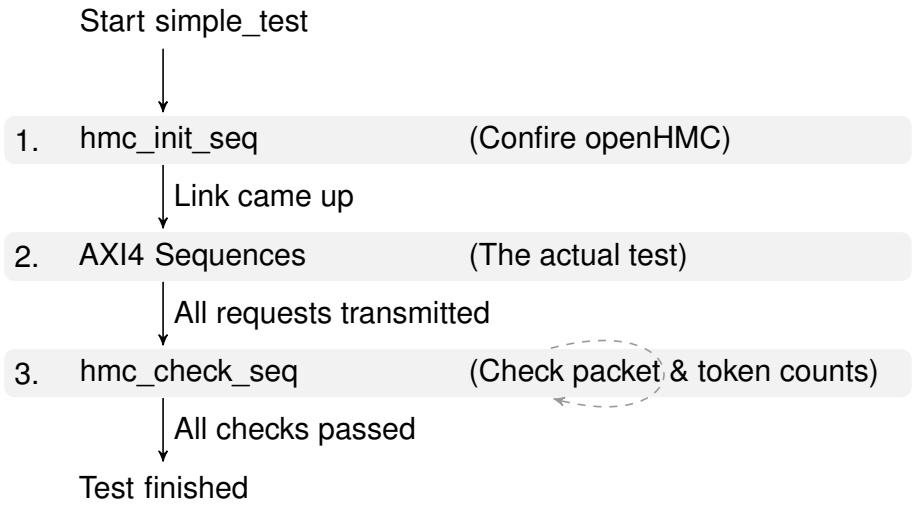

Figure 6.2: Test Procedure

packets. After all packets were transmitted on the AXI4 TX interface, the hmc_check_seq sequence is run. It ensures that all responses to non-posted requests were collected and that all tokens were successfully returned. If the test was successful, the performance counters for non-posted requests (sent_np) and received responses (rcvd_rsp) will be equal. Furthermore the HMC (hmc_tokens_remaining) and RX (rx_tokens_remaining) token count registers are periodically polled and compared against the initially distributed tokens. Default is 100 Tokens for both, HMC and openHMC. The test will abort and report a fatal error in case the hmc_check_seq is not completed successfully after approximately 200us.

6.5 F.A.Q.

Which simulators are supported ?

The openHMC testbench is tailored for the Cadence Incisive tool chain (NC Sim) version 13.10 and newer. Support for other simulators might be provided in the future.

Does the test support link retry ?

Not yet - Although the openHMC controller supports full and automatic link retry it is not supported by the test environment yet.

The test aborts / Warnings / Errors

Send an email to openhmc@ziti.uni-heidelberg.de and briefly describe the issue. Please attach the log file 'irun.log'

A » Acronyms

BFM	Bus Functional Model
CAG	Computer Architecture Group
CDR	Clock-Data Recovery
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	Loop-Up Table
MUX	Multiplexer
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

B » Register File Contents

Legend

HW Hardware access rights (through port list)

SW Software access rights (through RF interface)

wo write-only

ro read-only

rw read-write

Table B.1: Status General

Field	# Bits	Description & Encoding	Reset	HW	SW
link_up	1	Link is ready for operation	0	wo	ro
link_training	1	Link training in progress	0	wo	ro
sleep_mode	1	HMC is in Sleep Mode	0	wo	ro
lanes _reversed	1	0: Normal Operation 1: Lanes are reversed (lane 15/8 with 0, ...)	0	wo	ro
phy_rdy	1	SerDes reset is done	0	wo	ro
hmc_tokens _remaining	10	Amount of tokens remaining in the HMC input buffer	0	wo	ro
rx_tokens _remaining	10	Amount of tokens remaining in the MemCtrl RX input buffer	0	wo	ro
lane _polarity _reversed	NUM HMC LANES	0: Normal Operation 1: Data is logically inverted lane-by-lane	0	wo	ro

Table B.2: Status Init

Field	# Bits	Description & Encoding	Reset	HW	SW
lane _descramblers _locked	NUM HMC LANES	Lane by lane descrambler locked	0	wo	ro
descrambler_part _aligned	NUM HMC LANES	Lane by lane descrambler partially aligned	0	wo	ro
descrambler _aligned	NUM HMC LANES	Lane by lane descrambler fully aligned	0	wo	ro
all_descramblers _aligned	1	All descramblers are aligned	0	wo	ro
tx_init_status	2	Init Status of the TX Block 0: No init in progress 1: NULL1 2: TS1 3: NULL2	0	wo	ro
hmc_init_ts1	1	HMC sends TS1 packets	0	wo	ro

Table B.3: Performance Counter

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

Table B.4: Control

Field	# Bits	Description & Encoding	Reset	HW	SW
p_rst_n	1	Active low HMC reset.	1	ro	rw
hmc_init_cont_set	1	Allow descramblers to lock	1	ro	rw
set_hmc_sleep	1	Request HMC sleep mode. Sleep mode can be monitored by the 'sleep_mode' field in the Status General Register	1	ro	rw
scrambler _disable	1	Disable Scrambler and Descrambler for testing purposes	1	ro	rw
run_length _enable	1	Disable the run length limiter in the TX scrambler logic	1	ro	rw
first _cube_ID	1	Set the Cube ID of the first HMC connected. Used in irtry packets	0	ro	rw
debug _dont_send_tret	1	Prohibit memory controller from sending any TRET packets	0	ro	rw
rx_token _count	1	Set the input buffer space in the RX block	0x64	ro	rw
irtry_received _threshold	1	Set the number of irtry packets to be received until an action is performed.	0x10	ro	rw
irtry_to _send	1	Set the number of irtry to be sent	0x14	ro	rw
bit_slip _time	1	Set the time (in cycles) between to bit_slip impulses. Used for receiver alignment during initialization	0x24	ro	rw

Table B.5: Other Counter

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

C » Directory Structure

```
openhmc
├── export.sh
└── rtl
    ├── building_blocks
    │   ├── counter
    │   │   └── counter48.v
    │   └── fifos
    │       ├── async
    │       │   └── hmc_async_fifo.v
    │       ├── sync
    │       │   ├── sync_fifo_reg_stage.v
    │       │   ├── sync_fifo.v
    │       │   └── sync_fifos.f
    │       └── rams
    │           └── hmc_ram.v
    ├── hmc_controller
    │   ├── crc
    │   │   ├── crc_128_init.v
    │   │   └── crc_accu.v
    │   ├── register_file
    │   │   ├── hmc_controller_16x_rf.v
    │   │   └── hmc_controller_8x_rf.v
    │   ├── rx
    │   │   ├── rx_crc_compare.v
    │   │   ├── rx_descrambler.v
    │   │   ├── rx_lane_logic.v
    │   │   └── rx_link.v
    │   ├── tx
    │   │   ├── tx_crc_combine.v
    │   │   ├── tx_link.v
    │   │   ├── tx_run_length_limiter.v
    │   │   └── tx_scrambler.v
    │   ├── hmc_controller_top.f
    │   └── hmc_controller_top.v
    └── include
        └── hmc_field_functions.h
```

- sim
  - UVC
    - axi4_stream
      - sv
        - ...
    - cag_rgm
      - sv
        - ...
    - hmc_base_types
      - sv
        - ...
    - hmc_module
      - sv
        - ...
  - bfm
    - ..... PUT MICRON BFM HERE
  - tb
    - bfm
      - build
        - ...
      - src
        - behavioral
          - ...
        - rgm
          - openhmc
            - ...
        - targets
          - config.h ..... Configure the testbench
          - dut_openhmc_behavioral_bfm.sv ..... Top DUT
        - ...
      - testlib
        - simple_test
          - ...
        - ...
    - run
      - run.sh ..... RUN THIS FILE FOR SIMULATION
      - clean_up.sh ..... Cleanup old simulation files
      - run_files
        - ...
- doc
  - openhmc_doc_rev1_3

D » Revision History

1.3 The following changes have been made

Controller

- Fixed a misbehavior with the performance counters in tx_link
- Removed a redundant input register stage in rx_link to save resources and 1 cycle delay
- Minor modification in crc_accu to save resources and clear routing congestion

Testbench

- Added AXI-4 verification components and testbench to connect the Micron HMC BFM

Documentation (Section Number)

* Removed references to the HMC specification 2.0. The 30G-USR PHY / HMC specification 2.0 is not supported yet

6 Added new chapter for the Testbench

E » List of Figures

F » List of Tables

References

[1] Free Software Foundation, Inc. GNU Lesser General Public License. http://www.gnu.org/licenses/lgpl.html. [last accessed 12-Sep-2014].

[2] Hybrid Memory Cube Consortium. Hybrid Memory Cube Specification 1.1. http://www.hybridmemorycube.org/. [last accessed 12-Dec-2014].

[3] ARM Limited. AMBA AXI4-Stream Protocol Specification v1.0. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0051a/index.html. [last accessed 16-Aug-2014].