openHMC

a Configurable Open-Source Hybrid Memory Cube Controller

Computer Architecture Group, University of Heidelberg

in partnership with Micron Foundation

openHMC documentation Rev 1.4

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 configurable, 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. It further includes a test environment to evaluate the capabilities of the openHMC controller. The main objective of this project is to lower the barrier for others to experiment with the HMC, without the risks of using commercial solutions.

openHMC is licensed under the terms and conditions of version 3 of the Lesser General Purpose License[1].

Contact: openhmc@ziti.uni-heidelberg.de

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

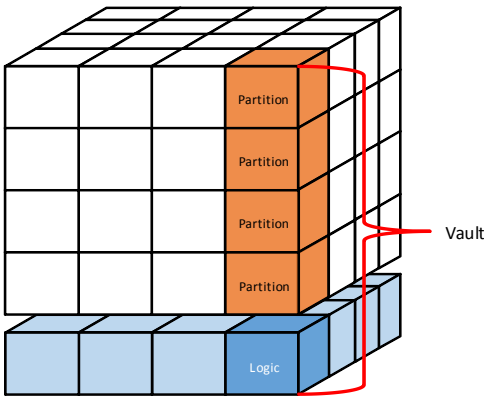

Figure 1.1: HMC: Abstract View

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.

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.

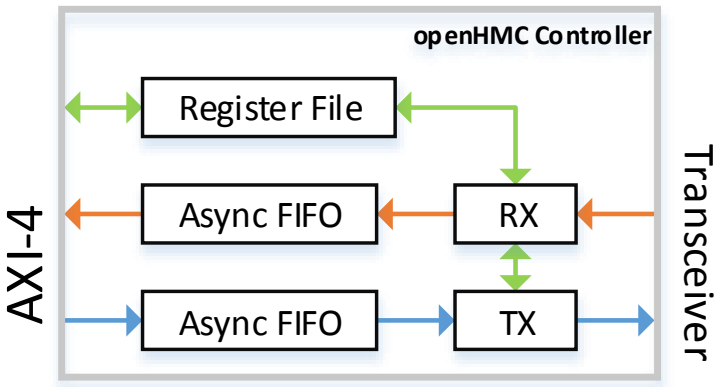

Figure 1.2: openHMC Host Controller Block Diagram

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 Configurations

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 configurations 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 (openhmc_top.v)

The openHMC top module instantiates and connects all logical sub-modules and does not contain any logic itself. It provides the AXI-4 , Transceiver and Register File interfaces. Figure 2.1 shows a more detailed view of the opeHMC controller top level including the two clock domains and main interface signals. For a full interface specification refer to Chapter 3. The host 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 (openhmc_async_fifo.v)

The asynchronous FIFOs connect the user logic in the clk_user clock domain to the openHMC controller in the clk_hmc clock domain. Both FIFOs appear as an AXI-4 Stream Protocol Interface to the user. The 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 using TAGs, if desired. Note that an unsupported command or a dln/lng mismatch may produce

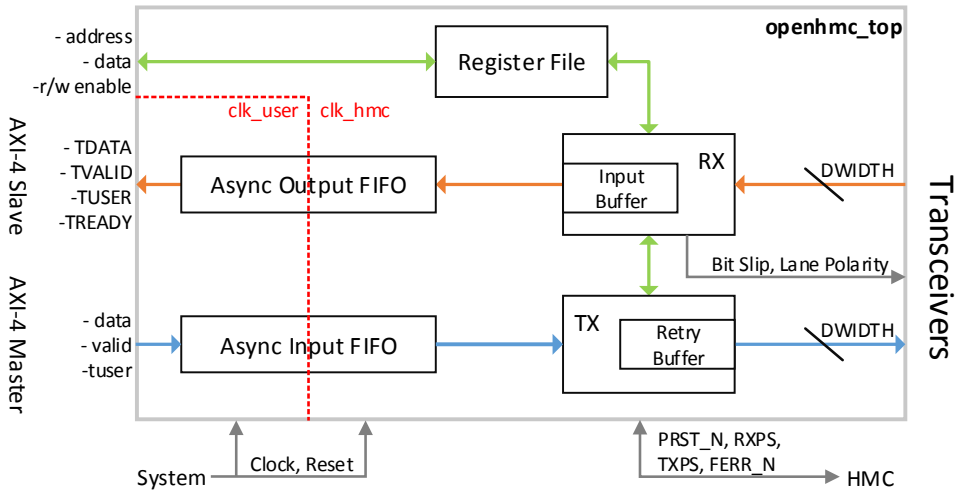

Figure 2.1: Detailed view of the openHMC Controller Top Module

undefined behavior in the current implementation. The 64bit tail must be set all to zero since it will be filled in the TX Link. Internally, the openHMC controller uses register stages to encapsulate logically-independent units, and to avoid critical paths due to excessive use of combinational logic. The main control function is implemented as the following Finite State Machine (FSM):

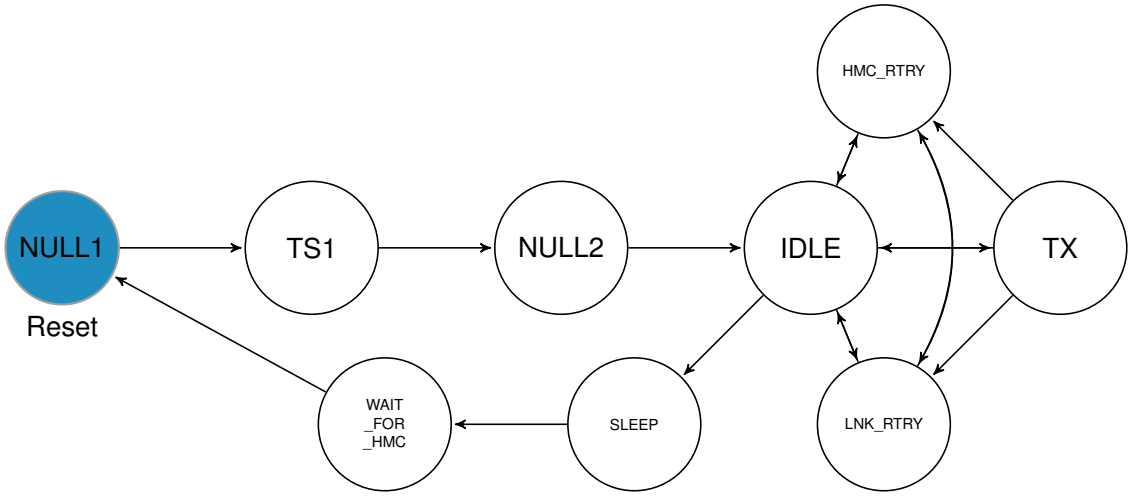

Figure 2.2: TX FSM

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 to exit sleep mode

Table 2.2: TX FSM Transition Table

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

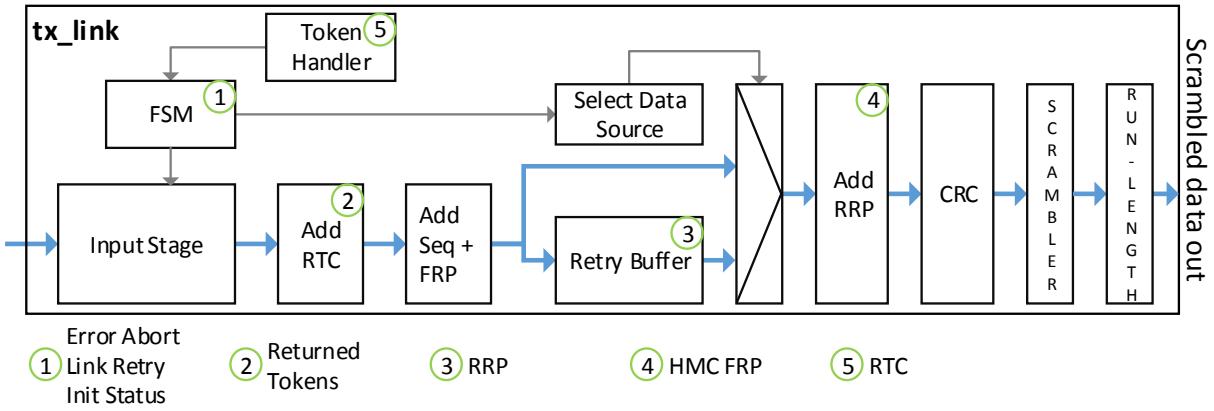

Figure 2.3: TX Link Diagram

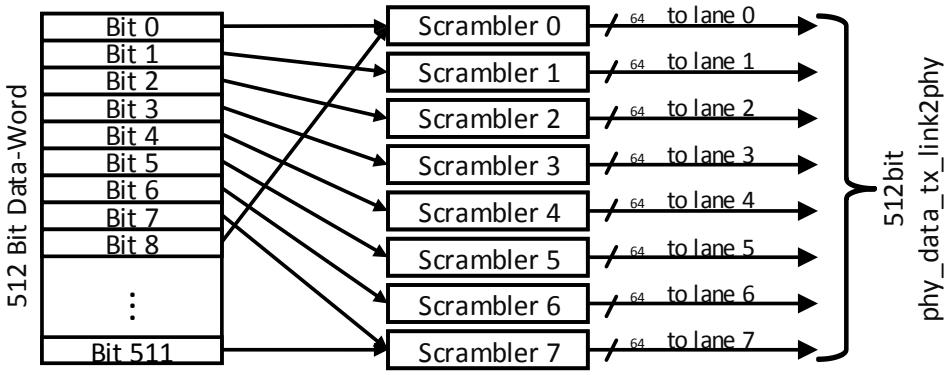

Figure 2.4: Data-Reordering: 4FLIT/512bit example

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 (openhmc_ram.v)

The retry buffer holds a copy of each FLIT transmitted for possible retransmission. NULL FLITs and flow packets, except TRET, are not subject to flow control and retransmission, and are therefore not saved in the retry buffer. The retry buffer actually consists of FPW

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

times 128-bit RAMs so that each FLIT can be addressed independently. One address, which is also the FRP is generated for each packet header. Since the required and accumulated RAM space is defined by the pointer size (FRP = RRP = 8 bit = 256 FLITs), the depth per RAM in this implementation is defined as 256 entries divided by FLITs per Word (FPW). 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.

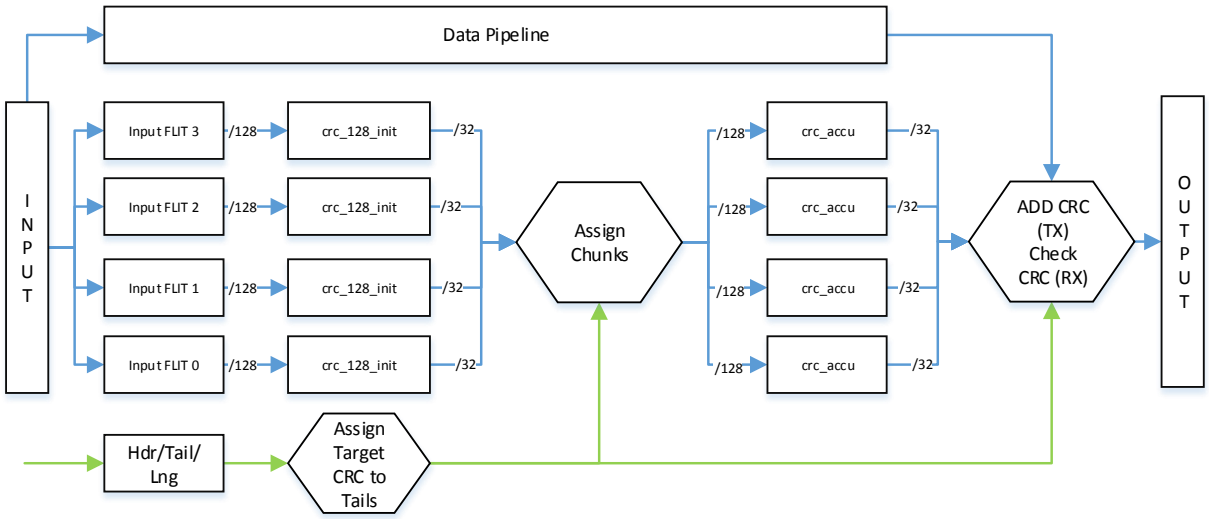

Figure 2.5: Scalable CRC Architecture: FPW=4 Example

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 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.

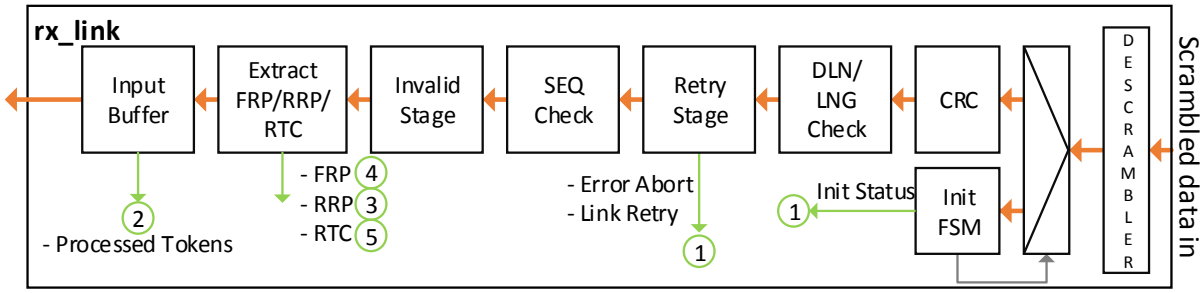

Figure 2.6: RX Link Diagram

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 (openhmc_sync_fifo.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. Note that the openHMC implementation does not forward poisoned packets to the input buffer.

2.5 Register File (openhmc_8x_rf.v and openhmc_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 the packet length or the CRC out of HMC headers or tails.

3 » Interface Description

This chapter contains an interface description for the top module openhmc_top.v. Due to the fact that the controller is configured using parameters many signal-widths depend on the configuration. The openhmc_top module contains a set of parameters that can be used to override the default configuration. All available parameters are listed in Table 3.1.

Table 3.1: Configuration Parameters

Parameter	Description	Default
LOG_FPW	Log of the desired data-width in FLITs	2
FPW	Desired data-width in FLITs (1FLIT = 128bit). Valid: 2/4/6/8	4
DWIDTH	FPW*128, width of the databus in bits	512
LOG_NUM_LANES	Log of the link width in lanes. Valid: 3/4	3
NUM_LANES	Link width in 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
CTRL_LANE_POLARITY	Set to 0 if lane polarity should be controlled by the transceivers or is not applicable	1
CTRL_LANE_REVERSAL	Set to 0 if lane reversal should be controlled by the transceivers or is not applicable	1
BITSLIP_SHIFT_RIGHT	Define how the parallel data is shifted by bit slip. Refer to the transceivers user guide	1
DBG_RX_TOKEN_MON	Enable/Disable monitoring of Tokens in the rx_link input buffer (1=enabled)	1

3.1 System Interface

The controller top module (openhmc_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 equal to or higher the frequency of clk_hmc. Therefore both clocks can

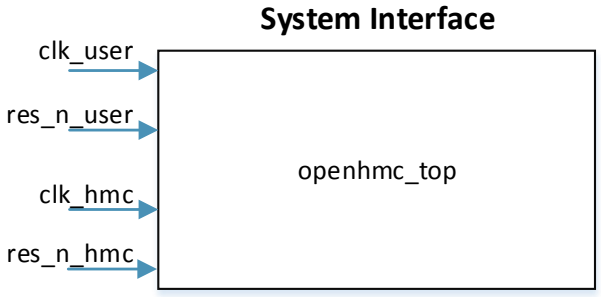

Figure 3.1: System Interface Diagram

origin from the same source. Figure 3.1 shows the system interface. Note that both resets are active low.

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'.

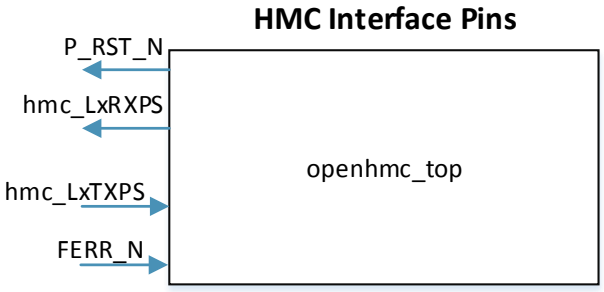

Figure 3.2: HMC Interface Pins Diagram

3.3 AXI-4 Stream Protocol Interface

The openHMC controller provides AXI-4 stream protocol interfaces for TX and RX. Both 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.

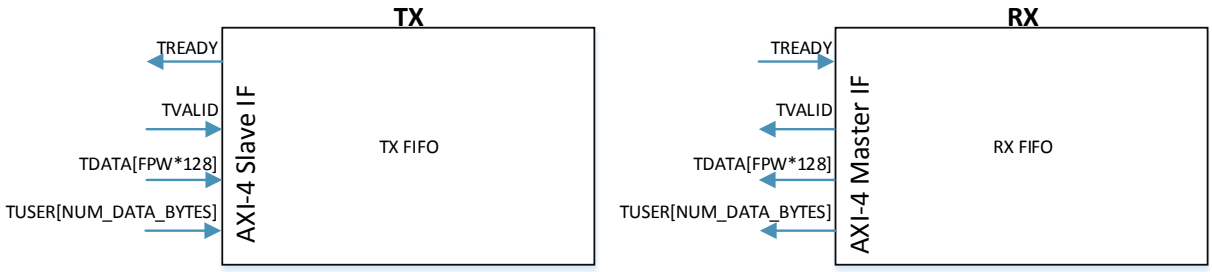

Figure 3.3: AXI-4 Interface Diagram

Note

The openHMC controller expects complete HMC packets at the TX interface, and outputs such at the RX interface. HMC request packets must be generated within the user logic, except the HMC packet tail which must be set to zero.

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.
- 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. Note that a single AXI cycle can carry (parts of) multiple packets on both interfaces, TX and RX. The user is responsible to populate all request header fields (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 the assertion of TREADY. When a response header appears, the packet does not need to be sampled consecutively throughout its tail.

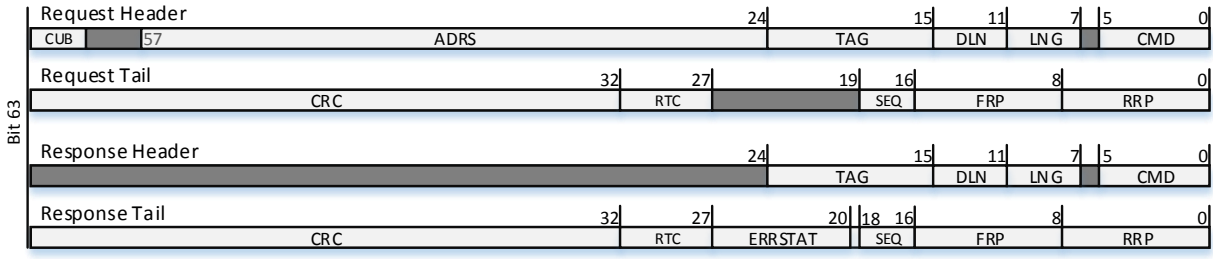

Figure 3.4: HMC Header and Tail

TUSER NUM_DATA_BYTES bit

The user is responsible to set the following information on the TX TUSER bus respectively the controller provides this information at the RX TUSER bus. Note that only a part of the TUSER bus is populated. That is 3*FPW bits on TX and 4*FPW on RX.

valid at TUSER index [FPW-1:0]: Valid FLIT indicator (including header and tail), one bit per FLIT

hdr at TUSER index [(2*FPW)-1:FPW]: Header indicator, one bit per FLIT

tail at TUSER index [(3*FPW)-1:2*FPW]: Tail indicator, one bit per FLIT

err_rsp [only on RX] at TUSER index [(4*FPW)-1:3*FPW]: Indicates an error response packet at the corresponding FLIT position. One bit per FLIT. Error response packets are single FLIT packets and have all flags (valid/hdr/tail/err_rsp) set

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

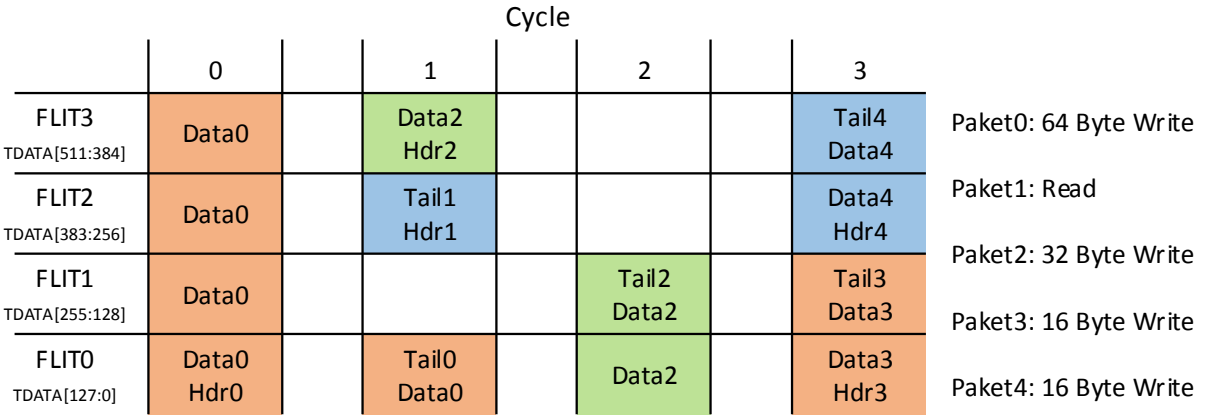

Figure 3.5: Example transactions on the AXI TX TDATA bus for FPW=4

	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

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

For proper operation of the interface, all FLITs of a packet on TX must be shifted in continuously without any 'bubble' FLITs or cycles in between. There is no constraint on 'bubbles'/NULL FLITs/NULL cycles between packets. However, TVALID on TX must **NOT** be set when there is no corresponding valid FLIT on TDATA / no valid bit set on TUSER. Additionally the frequency of the user clock clk_user driving the AXI-4 interface must be equal to or higher than clk_hmc. Due to the nature of the asynchronous FIFO that is used, empty and full signals may be delayed and might cause misbehavior in the tx_link.

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 on the parallel input data during initialization. Connect these to the corresponding transceiver. If lane_polarity is performed within the transceivers, the phy_lane_polarity output must be used. 'CTRL_LANE_POLARITY' must be set to 1 in this case. For CTRL_LANE_POLARITY=0 phy_lane_polarity is tied to 0. All signals are summarized in Table 3.2. 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
phy_bit_slip	NUM_ LANES	Bit_slip is used to compensate lane to lane skew. Bit_slip is controlled by the rx_link for each lane individually
phy_lane_polarity	HMC_ NUM_ LANES	Connect transceiver polarity inputs if polarity is controlled within the transceivers and 'CTRL_LANE_POLARITY' is set to 1.

Listing 3.1: Transceiver Connectivity Example for FPW=4 and NUM_LANES=8

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

openhmc_top #(...parameter list...) openhmc_I (
        ⋮
        .phy_data_tx_link2phy(tx_data),
        .phy_data_rx_phy2link(rx_data),
        .phy_bit_slip(rx_bit_slip),
        .phy_lane_polarity(rx_lane_polarity),
        ⋮
        );
```

```
transceiver_top #(...) transceiver_I (
        ⋮
        .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]),
        ⋮,
        .lane0_polarity_in(rx_lane_polarity[0]),
        .lane1_polarity_inrx_lane_polarity[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 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

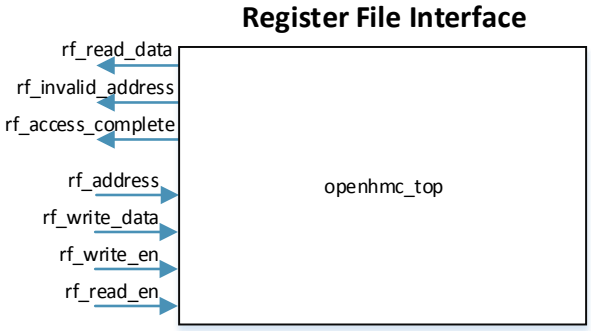

Figure 3.7: Register File Interface Diagram

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
error_abort_not_cleared	0xC	Number of error_abort_mode not cleared

address 0x10. Refer to Table 3.4 for the address mapping. For a full listing of all fields within the RF see Appendix B.

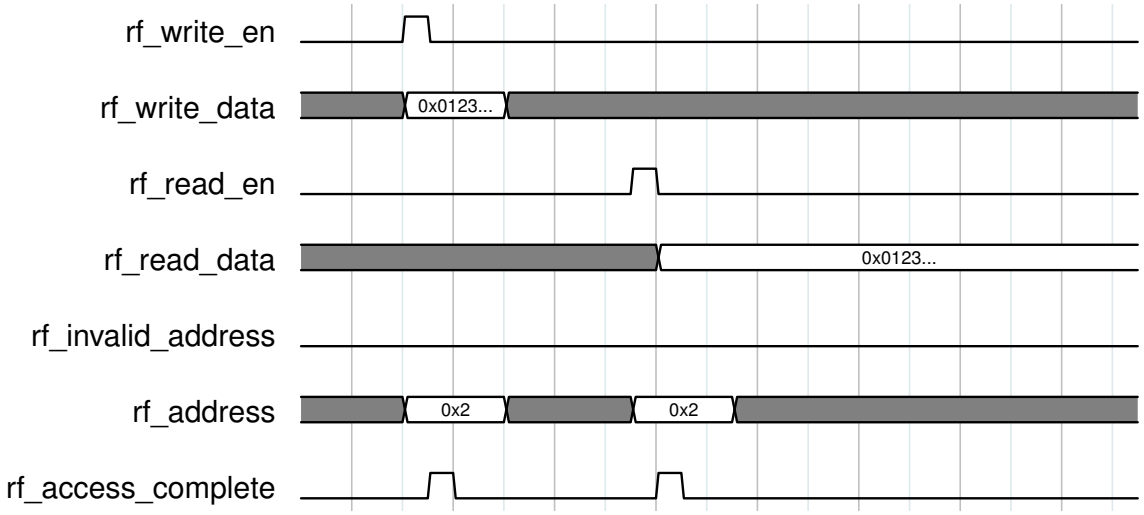

Figure 3.8: Register File Access: Write and read register 0x2

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 implemented 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 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

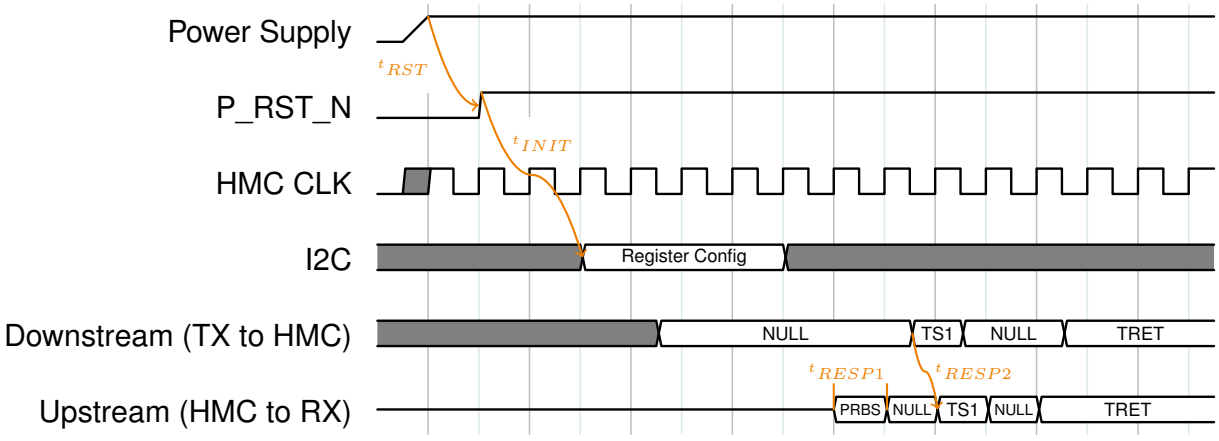

Figure 4.1: TX-Link: Initialization Timing

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.

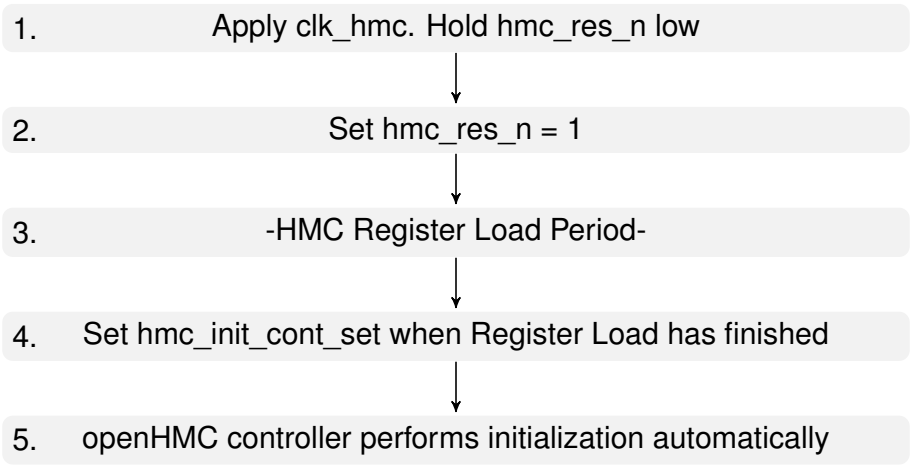

Figure 4.2: openHMC Controller Power Up Steps

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 completed and tx_link 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.

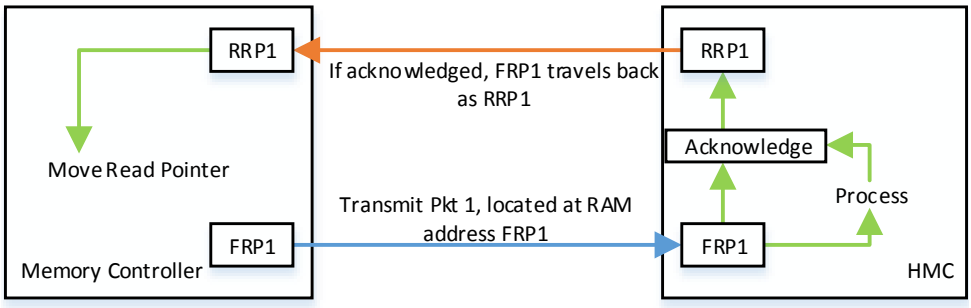

Figure 4.3: Pointer Flow

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.

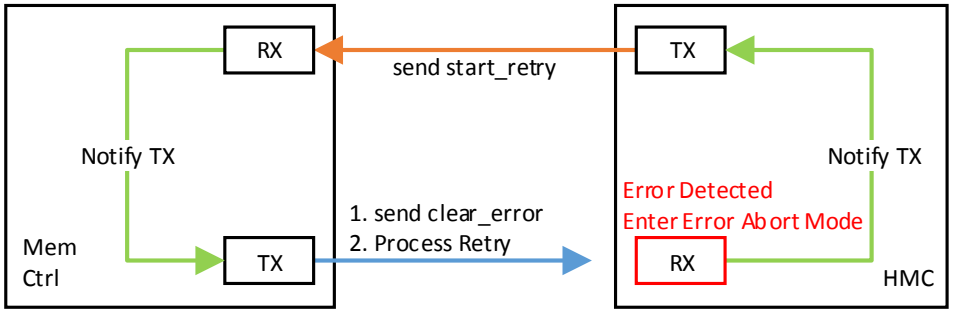

Figure 4.4: TX Link Retry

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.

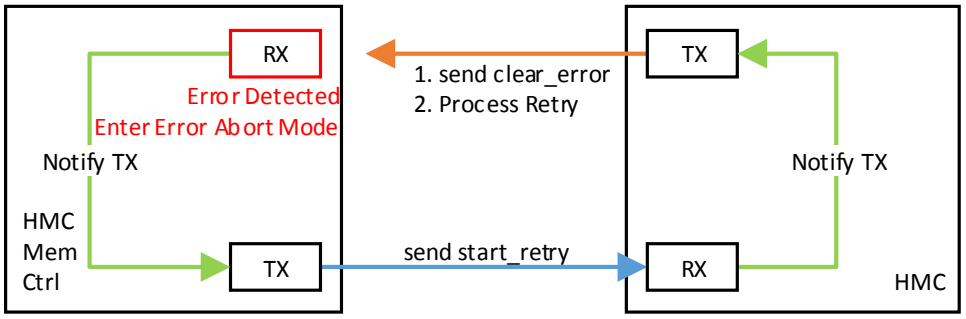

Figure 4.5: HMC Retry

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 Retry Pointer Loop Time

According to the HMCC specification[2], the retry pointer loop time should not exceed certain limitations. These limitations vary depending on the selected link speed (10Gbit/s, 12Gbit/s, or 15Gbit/s). Factors such as the HMC delay, host delay, serialization, and de-serilazation contribute to the total retry pointer loop time. In case the host exceeds the maximum allowable delay, the HMC retry buffer may run full and therefore throttle packet streaming which leads to NULL FLITs between transaction packets. Table 4.2 lists both, internal HMC delay and host allowable delay in nanoseconds. It is based on the assumption that the retry buffer full period is approximately as twice as big when running a link at half-width (8 lane). However, this is not mentioned in the HMCC specification. Note that all calculations in this section were performed with the run length limiter deactived (HMC_RX_AC_COUPLED=0) and no lane polarity control (CTRL_LANE_POLARITY=0).

4.6.1 TX Link Retry Pointer Delay

Table 4.3 shows the worst case delay for the RRP (the former HMC FRP) to be embedded, starting at the point where the RRP becomes available at the tx_link input until the RRP appears in the scrambled output data stage. The best case delay for RRP embedding occurs when the RRP becomes available right in a cycle where a tail is processed in the 'Add

Table 4.2: Retry Pointer Loop Time

Lane Speed [Gb/s]	Lanes	Retry Buffer Full Period [ns]	HMC Delay[ns]	Max Host Delay[ns]
10	8	307.2	26.5	280.7
12.5	8	327.6	25.9	301.7
15	8	272.8	22.3	250.5
10	16	153.6	26.5	127.1
12.5	16	163.8	25.9	138
15	16	136.4	22.3	114.2

Table 4.3: TX Link Worst Case RRP Embed Delay

DWIDTH [FLITs]	2 FLIT (256Bit)		4 FLIT (512Bit)		6 FLIT (768Bit)		8 FLIT (1024Bit)	
Stage	Cycles	Acc	Cycles	Acc	Cycles	Acc	Cycles	Acc
Add RRP	5	5	3	3	3	3	2	2
CRC	4	9	4	7	4	7	4	6
Scrambler	1	10	1	8	1	8	1	7
Max Delay[cycles]*	10		8		8		7	

*Max Delay increases by 1 cycle if the Run Length Limiter is used (HMC_RX_AC_COUPLED=1)

Table 4.4: RX Link RRP Process/Extract Delay

DWIDTH [FLITs]	2 FLIT (256Bit)		4 FLIT (512Bit)		6 FLIT (768Bit)		8 FLIT (1024Bit)	
Stage	Cycles	Acc	Cycles	Acc	Cycles	Acc	Cycles	Acc
Descrambler	1	1	1	1	1	1	1	1
to CRC	1	2	1	2	1	2	1	2
CRC	4	6	4	6	4	6	4	6
DLN/LNG	1	7	1	7	1	7	1	7
Retry	1	8	1	8	1	8	1	8
Seq	1	9	1	9	1	9	1	9
Invalidation Stage	5	14	3	12	3	12	2	11
Extraction	1	15	1	13	1	13	1	12
Total Delay[cycles]*	15		13		13		12	

*Delay increases by 1 cycle if CRTL_LANE_POLARITY=1

RRP' stage. In the worst case, a new packet has just begun at the top-most FLIT position. Embedding will therefore be delayed by the number of cycles it takes to forward a packet until its tail is seen at this stage. Hence, the maximum delay is measured with a 128-Byte request in 2-FLIT (256Bit) configuration where the RRP becomes available and FLIT(0) is NULL, FLIT(1) the header of the packet. As can be seen in Table 4.3, the worst case delay reduces with wider data-paths.

4.6.2 RX Link Retry Pointer Delay

Table 4.4 shows the delay for the HMC FRP to be extracted and passed to the TX Link, starting at the point where the HMC FRP becomes available at the scrambled_data_in input until it was extracted and becomes available for the TX Link to be embedded. The delay for HMC FRP extraction decreases as the datapath becomes wider, since the depth of the 'Invalidation Stage' decreases.

Table 4.5: Combined Retry Pointer Delay

DWIDTH [FLITs]	2 FLIT (256Bit)	4 FLIT (512Bit)	6 FLIT (768Bit)	8 FLIT (1024Bit)
TX	10	8	8	7
RX	15	13	13	12
Total Delay[cycles]	25	21	21	19

4.6.3 Combined Retry Pointer Loop Time

Table 4.5 summarizes the results of openHMC TX and RX pointer delays (Table 4.3 and Table 4.4).

4.7 openHMC Configuration

According the configuration of the data-width (DWIDTH), half-width or full-width (NUM_LANES) and their respective lane speed, Table 4.6 lists selected configurations that can be applied. Table 4.7 lists all valid parameter sets. The resulting core clocking frequency clk_hmc is calculated with:

$$\text{clk\_hmc}[\text{MHz}] = \frac{\text{NUM\_LANES} * \text{LANE\_SPEED}[\text{Gbit/s}]}{\text{DWIDTH} * 10^6}$$

Table 4.6 furthermore summarizes the results for the retry pointer loop time through the openHMC controller. Refer to Table 4.2 for the maximum allowed host delay It can be seen that not all configurations stay within the host allowable delay. Note that additional delay will be introduced through serialization and de-serialization.

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. The top level parameter LOG_MAX_RTC must be set accordingly, i.e. the actual token count must be equal to or less than $2^{LOG\_MAX\_RTC}$.

4.8 HMC Configuration

Maximum Packet Size

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

Table 4.6: Example Configurations

DWIDTH [bit]	NUM _LANES	lane speed [Gbits]	clk_hmc [MHz]	Period [ns]	Worst Case Delay [cycles]	Worst Case Delay[ns]
256	8	10	312.5	3.2	25	80
256	8	12.5	390.625	2.56	25	64
512	8	10	156.25	6.4	21	134.4
512	8	12.5	195.3125	5.12	21	107.52
512	8	15	234.375	4.27	21	89.6
512	16	10	312.5	3.2	21	67.2
512	16	12.5	390.625	2.56	21	53.76
768	8	15	156.25	6.4	21	134.4
768	16	10	208.33	4.8	21	100.8
768	16	12.5	260.417	3.84	21	80.64
768	16	15	312.5	3.2	21	67.2
1024	16	10	156.25	6.4	19	121.6
1024	16	12.5	195.3125	5.12	19	97.28
1024	16	15	234.375	4.27	19	81.06

Table 4.7: Valid parameter sets

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

HMC Token Count

To avoid misbehavior for any of the listed configurations, set the token count within the HMC token register to at least 25'd

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 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 configurations listed in Table 5.1. The Xilinx Vivado Design Suite 2014.3.1 was used as implementation tool. All runs were performed with the configuration parameters set as listed in Table 5.2.

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
2	4	8	12.5	195.3125	Xilinx Virtex Ultrascale XCVU095
3	2	8	10	312.5	Xilinx Virtex Ultrascale XCVU095
4	2	8	12.5	390.625	Xilinx Virtex Ultrascale XCVU095

Table 5.2: Top-Level Implementation Parameters

Parameter	Value
LOG_MAX_RTC	8
HMC_RX_AC_COUPLED	1
CTRL_LANE_POLARITY	1
CTRL_LANE_REVERSAL	0
DBG_RX_TOKEN_MON	0

In addition to the FPGA-verified configurations, the openHMC controller was successfully implemented in various other configurations such as full-width @ 12.5Gbps.

5.2.1 Resource Utilization

Table 5.3 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 approximate and may slightly vary for different implementation strategies.

5.3 Optimization Techniques

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

Table 5.3: Resource Utilization

ID	LUTs combined	Registers	BRAM B36/B18
1,2	16170	15692	30
3,4	8200	9334	8

Disable Run Length Limiter

If Controller TX to HMC RX is DC coupled, set the parameter HMC_RX_AC_COUPLED in the top file 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.

Disable Lane Polarity/Lane Reversal control

Both, lane polarity and lane reversal can either be applied in the controller or the transceivers. While disabling lane reversal in the controller only saves resources, disabling lane polarity also reduces the latency through rx_link. The corresponding parameters to set are 'CTRL_LANE_POLARITY' and 'CTRL_LANE_REVERSAL. Refer to Chapter 3.

Disable rx_link Input Buffer Token Monitoring

For debugging purposes, the remaining rx_link input buffer space (in Tokens) is counted in the tx_link. To save resources, monitoring can be disabled by setting the parameter DBG_RX_TOKEN_MON = 0 in the top file (refer to Table 3.1) .

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

The Universal Verification Methodology (UVM) based test environment can be used to demonstrate and verify the functionality of the openHMC controller. It is designed following the IEEE Standard for SystemVerilog[4] and tested for the Cadence Incisive tool chain (NC Sim) version 14.10 and newer. Other simulators might be supported in the future. The Micron BFM of the HMC is required and can be obtained under NDA. Please contact openhmc@ziti.uni-heidelberg.de for more information.

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/tb/run and execute run.sh by typing './run.sh' for example. Table 6.1 lists all available arguments. A test in an 2FLIT and 16lanes configuration with high UVM verbosity may be started with:

./run.sh -f 2 -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. 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
-o		Enable Coverage
-s	X	Start the test with a different seed
-t	X	Specify a test (see Section 6.5)
-v	X	Verbosity of the debug output. Available values are UVM_NONE, UVM_LOW (default), UVM_MEDIUM, and UVM_HIGH
-?		Print usage help

6.3 Test Environment

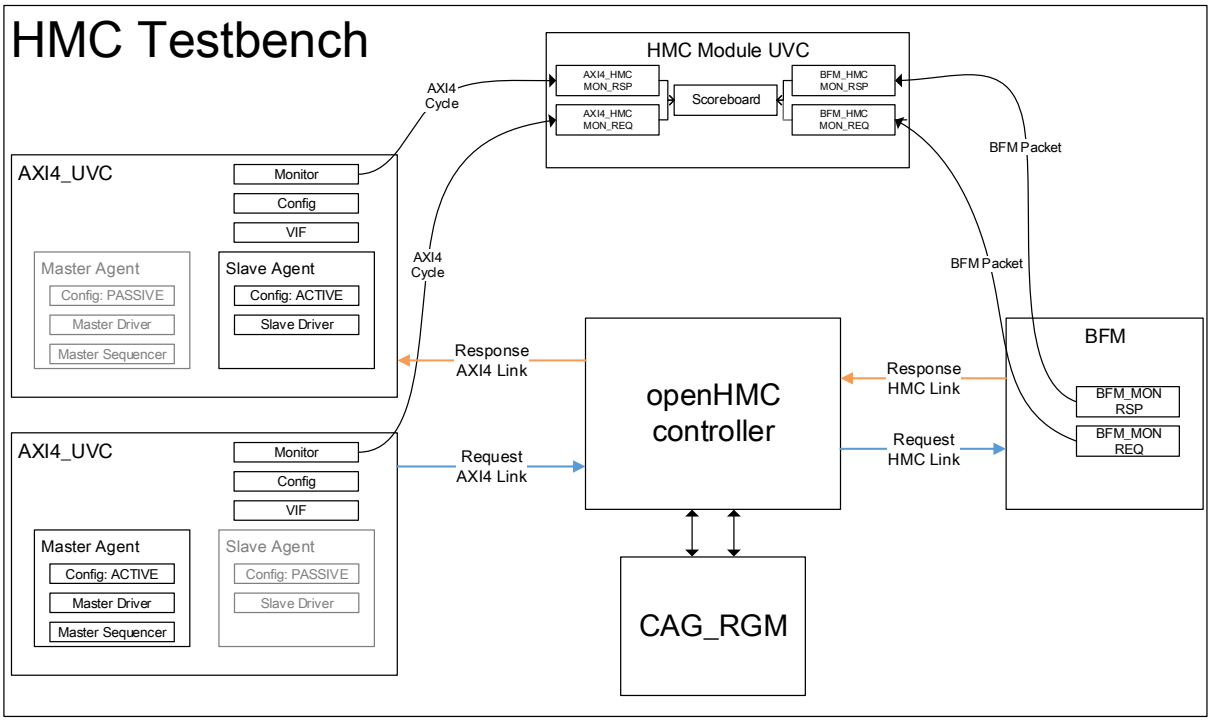

Figure 6.1: HMC Testbench

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

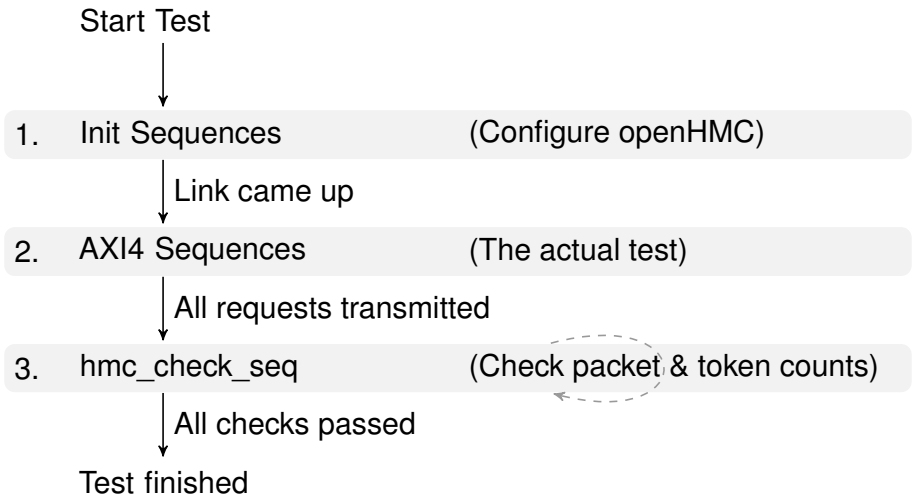

Figure 6.2: Test Procedure

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 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)

6.3.1 Randomization

Features such as lane reversal, lane polarity, lane delay, and HMC and openHMC token counts are randomized and can be user-constrained in 'hmc_link_config.sv'.

6.4 Test Procedure

All tests are processed in three phases as shown in Figure 6.2. After the test started, BFM and the openHMC controller are configured during the bfm_init_seq and hmc_init_seq , respectively. After the link came up (signalized by the link_up bit in the register file 'status register'), the actual test is started. Depending on the test, one or more hmc_pkt_sequences are executed. These sequences will execute one or multiple hmc_2_axi4_sequence with additional constrains. Finally, after the actual test has finished, a check sequence

(hmc_check_sequence) is executed. This sequence ensures that all responses to non-posted requests were collected and that all tokens were successfully returned. The test will abort and report a fatal error in case the hmc_check_seq is not completed successfully after approximately 200us.

6.5 The Tests

The following tests are available:

simple_test (default)
: This test sends a small amount ($<= 250$) of unconstrained packets.

small_pkt_test
: Constrain packets to be small (max length = 2 Flits).

big_pkt_test
: Constrain packets to be large (min length = 6 Flits).

posted_pkt_test
: Constrain packets to be posted. This test randomly selects one of the test sequences described above. Repeat max 40times.

non_posted_pkt_test
: Constrain packets to be non posted. This test randomly selects one of the test sequences described above. Repeat max 40times.

atomic_pkt_test
: Constrain packets to be atomic. Repeat max 40times.

init_test
: Runs initialization including TRET exchange. No data packets will be sent.

sleep_mode
: Send a packet sequence, enter sleep, exit sleep, (repeat). Use of sleep mode is experimental. Warnings that might be thrown by the BFM (violation of timing tOP) can be safely ignored.

high_delay_pkt_test
: Constrain the delay between packets to be above 90 FLITs.

zero_delay_pkt_test
: Constrain the delay between packets to be zero.

small_pkt_hdelay_test

This test combines the constrains of small_pkt_test and hdelay_pkt_test.

small_pkt_zdelay_test

This test combines the constrains of small_pkt_test and zdelay_pkt_test.

big_pkt_hdelay_test

This test combines the constrains of big_pkt_test and hdelay_pkt_test. The Packet size is constraint to be above 6 Flits with high packet delay.

big_pkt_zdelay_test

This test combines the constrains of big_pkt_test and zdelay_pkt_test. The Packet size is constraint to be above 6 Flits with zero packet delay.

6.6 Error Injection / Link Retry

Automatic, randomized error injection on both directions of the link can be configured in hmc_link_config.sv. At the time of writing, the latest revision of the BFM is 28965. For this revision, error injection in response packets can be used without any limitations (cfg_rsp_* in hmc_link_config.sv). CAG does not recommend error injection in request packets due to issues with the BFM. However, request packet error injection may be used when sequence number poisoning is left out (cfg_req_seq=0 in hmc_link_config.sv !).

6.7 F.A.Q.

Which simulators are supported ?

The openHMC testbench is tailored for the Cadence Incisive tool chain (NC Sim) version 14.10 and newer. Support for other simulators might be provided in the future.

Does the test support link retry ?

Yes - link retry is supported. The latest BFM (at the time of writing: revision 28965) is recommended.

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

Note that some field-widths depend on the parameter NUM_LANES. All bits that are not listed are reserved and tied to logical 0.

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	Bit	Width [Bits]	Description & Encoding	Res	HW	SW
link_up	0	1	Link is ready for operation	0	wo	ro
link_training	1	1	Link training in progress	0	wo	ro
sleep_mode	2	1	HMC is in Sleep Mode	0	wo	ro
lanes _reversed	3	1	0: Normal Operation 1: Lanes reversed (lane 15/8 with 0, ...)	0	wo	ro
phy_ready	8	1	SerDes reset is done	0	wo	ro
hmc_tokens _remaining	25:16	10	Number of Tokens remaining in the HMC input buffer	0	wo	ro
rx_tokens _remaining	41:32	10	Number of Tokens remaining in the rx_link input buffer (if param DBG_RX_TOKEN_MON = 1)	0	wo	ro
lane _polarity _reversed	55:48(8x) 63:48(16x)	NUM LANES	0: Normal Operation 1: Data is logically inverted lane-by-lane	0	wo	ro

Table B.2: Status Init

Field	Bit	Width [Bits]	Description & Encoding	Res	HW	SW
lane _descramblers _locked	7:0(8x) 15:0(16x)	NUM LANES	Lane by lane descrambler locked	0	wo	ro
descrambler _part _aligned	23:16(8x) 31:16(16x)	NUM LANES	Lane by lane descrambler partially aligned	0	wo	ro
descrambler _aligned	40:32(8x) 47:32(16x)	NUM LANES	Lane by lane descrambler fully aligned	0	wo	ro
all_descramblers _aligned	48	1	All descramblers are aligned	0	wo	ro
tx_init_status	50:49	2	Init Status of the TX Block 2'b00: No init in progress 2'b01: NULL1 2'b10: TS1 2'b11: NULL2	0	wo	ro
hmc_init_ts1	51	1	HMC sends TS1 packets	0	wo	ro

Table B.3: Other Counter (Each Entry equals one Register)

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

Table B.4: Performance Counter (Each Entry equals one Register)

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.5: Control

Field	Bit	Width [Bits]	Description & Encoding	Res	HW	SW
p_rst_n	0	1	Active low HMC reset	0	ro	rw
hmc_init_cont_set	1	1	Allow descramblers to lock	0	ro	rw
set_hmc_sleep	2	1	Request HMC sleep mode. Sleep mode can be monitored by the 'sleep_mode' field in the Status General Register	0	ro	rw
scrambler _disable	3	1	Disable Scrambler and Descrambler for testing purposes	0	ro	rw
run_length _enable	4	1	Disable the run length limiter in the TX scrambler logic	1	ro	rw
first _cube_ID	7:5	3	Set the Cube ID of the first HMC connected. Used in irtry packets	0	ro	rw
debug _dont_send_tret	8	1	Prohibit controller from sending any TRET packets	0	ro	rw
debug _halt_on _error_abort	9	1	HALT tx_link after rx_link entered error abort	0	ro	rw
debug _halt_on _tx_retry	10	1	HALT tx_link after performing a retry	0	ro	rw
rx_token _count	25:16	10	Set the input buffer space in the RX block	0x64	ro	rw
irtry_received _threshold	36:32	5	Set the number of irtry packets to be received until an action is performed	0x10	ro	rw
irtry_to _send	44:40	5	Set the number of irtry to be sent	0x16	ro	rw
bit_slip _time	53:48	6	Set the time (in clk_hmc cycles) between to bit_slip impulses. Used for receiver alignment during initialization. Value is based on transceiver requirements	0x28	ro	rw

C » Directory Structure

```
openhmc
├── export.sh
└── rtl
    ├── building_blocks
    │   ├── counter
    │   │   └── counter48.v
    │   └── fifos
    │       ├── async
    │       │   └── openhmc_async_fifo.v
    │       ├── sync
    │       │   ├── openhmc_sync_fifo_reg_stage.v
    │       │   ├── openhmc_sync_fifo.v
    │       │   └── openhmc_sync_fifos.f
    │       └── rams
    │           └── openhmc_ram.v
    ├── hmc_controller
    │   ├── crc
    │   │   ├── crc_128_init.v
    │   │   └── crc_accu.v
    │   ├── register_file
    │   │   ├── openhmc_16x_rf.v
    │   │   └── openhmc_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
    │   ├── openhmc_top.f
    │   └── openhmc_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
        - rgm
        - serdes
          - behavioral
        - targets
          - dut_openhmc_behavioral_bfm.sv ..... Top DUT
        - hmc_link_config.sv ..... Test configuration
      - testlib
        - simple_test
          - ...
          - ...
        - ...
      - run
        - run.sh ..... RUN THIS FILE FOR SIMULATION
        - clean_up.sh ..... Cleanup old simulation files
        - run_files
          - ...
- doc
  - openhmc_doc_rev1_4

D » Revision History

1.4 The following changes have been made

Controller

- Improved CRC modules for relaxed timing and siginificant resource savings
- Added additional parameters to allow greater control. Refer to Chapter 3 for more information
- TX_Link: Fixed retry buffer overflow condition
- RX_Link: Several fixes to issues that caused link retry to fail
- RX_Link: Poisoned packets do not enter the input buffer. Tokens are returned beforehand
- RX_Link: Renamend some nets in rx_link and added dedicated initialization datapath to relax timing
- RX_Link: Fixed initialization issue associated with 2FLIT/full-width config where descrambler won't align
- RX_Link AXI IF: Added error response packet indicator on TUSER bus
- rx_crc_compare: Added check for reserved '0' fields within flow packets
- Added debug register 'error_abort_not_cleared' to monitor failed link retries after the retry timeout period
- Added debug registers 'debug_halt_on_error_abort' and 'debug_halt_on_tx_retry' to freeze tx_link after either event occured

Testbench

- Major update (see Chapter 6)

Documentation (Section Number)

2.4.3 Added a note that poisoned packets do not enter the input buffer

3.3 Added a note that empty cycles are not allowed on TX when TVALID=1

3.3 Added error response indicator for RX AXI interface

4 Added retry pointer loop time section

5 Added new example implementations

6 Major update

B Added new fields to control registers. Corrected wrong reset values

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].

[4] IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group. IEEE Std 1800-2012, IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language. Technical report, Feb. 21, 2013.