HMC Controller IP

User Guide

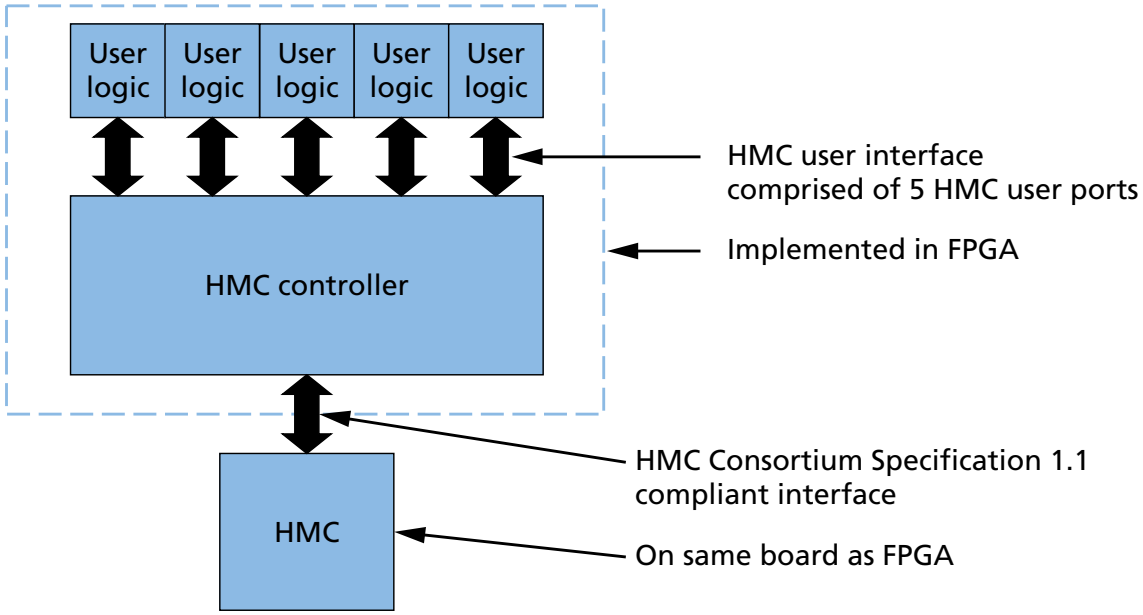
Features

- HMC specification 1.1 compliant
  - Half- and full-width modes: 8 and 16 lanes
  - HMC device bring-up and power state management
  - 16-byte to 128-byte R/W requests + Atomics
  - Fault tolerant
  - Test, debug and characterization features are built-in
  - Lightweight, multiported user interface
- Device support: IP integration for Xilinx and Altera FPGA
- ASIC IP available upon request
- Design example: HPCC Random Access Benchmark (GUPS)
- Evaluation platforms
  - Micron SB-800
  - Micron AC-510

Introduction

This document is intended to describe the use of the HMC controller, which talks directly to a Micron HMC device. We provide FPGA-based evaluation systems, with an HMC and an FPGA laid down on the same board. We also provide a design example to highlight the power of the HMC with this controller.

Figure 1: HMC Controller Block Diagram



HMC Contoller Signals

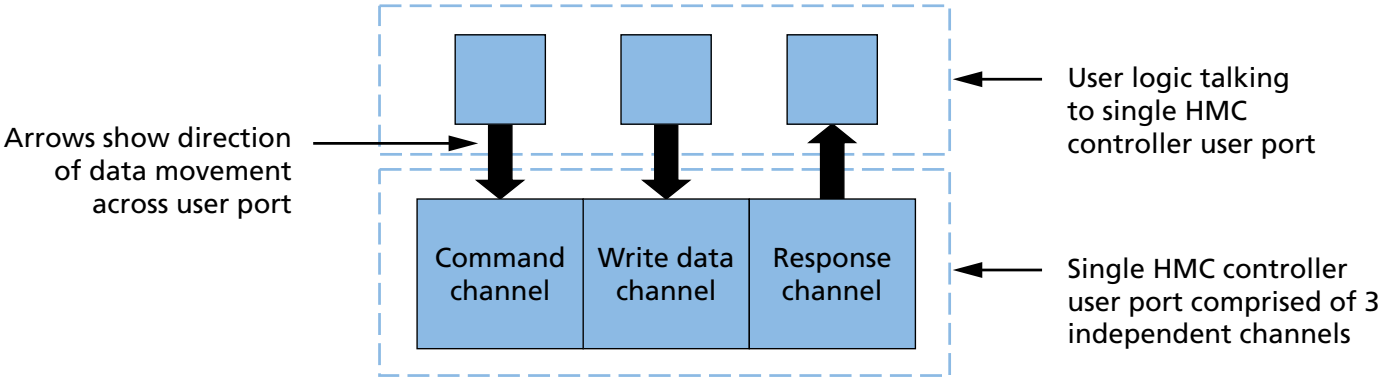
Table 1: Clock, Reset, and Global Signals

Signal Name	Type	Width (Bits)	Description	Notes
tx_clk	Output	1	Clock from the TX transceivers.	1
rx_clk	Output	1	Clock from the RX channel.	1
rst	Output	1	Controller is in reset when this is asserted (asynchronous).	–
trained	Output	1	Controller link has been brought up when this is asserted (asynchronous).	–

Notes:
1. Clocks tx_clk and rx_clk run at the same speed on each specific board: 250 MHz on SB-800 and 187.5 MHz on AC-510. Note that while the clocks run at the same speed, they may be out of phase with respect to one another; proper clock crossing techniques must be used when crossing from one clock domain to the other.
2. Clocks tx_clk and rx_clk are available to the user, but their use is optional.

Single and Multiple User Port Signals

Figure 2: Channel Block Diagram



This interface is very similar to the AXI protocol, including essential components of that protocol without unnecessary details. Highlights include:

- Three independent channels are supported: Command, write data, response data.
- The first two channels have their own valid-ready handshake, like the valid-ready handshake of the AXI protocol: Data moves only when both valid and ready are asserted.
- The response channel has only a valid signal, so the user logic cannot provide any back pressure; the user logic must not send a command to the controller if it cannot accept a returning controller response.
- Part of the command channel includes a tag field needed primarily for commands producing a response; responses may come back out-of-order and the user logic reorders the responses as required.
- Both READ and WRITE commands should be sent to the controller on the same channel, the command channel. This enables the user logic to specify its own priority.
- The write data channel is very similar to a FIFO interface. The expectation is that most users will drive this directly from the output of a FIFO.

The controller exposes five ports to the user logic. The multiple user ports extend the single user interface to multiple ports by appending a suffix with the port number to all signals in the table below. For example, cmd becomes cmd_p0 for port 0, cmd_p1 for port 1, and so on.

The controller automatically arbitrates (see note below) between these user ports. It also routes responses back to the originating user port. Note that Micron does not provide any guarantees about command ordering for commands that are issued on different user ports, because this is very application-dependent.

Note: The arbiter used is a fixed-schedule, round-robin arbiter, where the link bandwidth is evenly divided between these user ports. To maximize the controller's efficiency, users should distribute requests across many user ports.

Table 2: Port Signals

Signal Name	Type	Width (Bits)	Description
clk	Input	1	User-driven clock for the HMC interface for this user port. All command, write data, and response channels are synchronous to this clock. Minimum supported clock frequency is equal to PicoClk, and the maximum supported frequency is equal to tx_clk.
Command Channel			
cmd_valid	Input	1	Valid bit for the command channel. Do not assert this until you are prepared to send a command.
cmd	Input	4	Command field.
addr	Input	34	Address field. This is a byte-addressable memory and must be 16B aligned.
size	Input	4	Size (in 16B words) of the READ or WRITE request. Valid range = 1 to 8 word(s).
tag	Input	6	Tag field. Mainly used to match a response to its original command.
cmd_ready	Output	1	Ready bit for the command channel. Command is sent to the controller when both valid and ready are asserted.
Write Data Channel			
wr_data_valid	Input	1	Valid bit for the write data channel. Do not assert this until you are prepared to send another beat of data.
wr_data	Input	128	Write data.
wr_data_ready	Output	1	Ready bit for the write data channel. Write data is accepted by the controller when both valid and ready are asserted.
Response Channel			
rd_data_valid	Output	1	Valid bit for the read data channel. User logic must be ready to accept this data when the valid bit is asserted.
rd_data	Output	128	Read data.
rd_data_tag	Output	6	Tag field. Used to match a response to the original command (if required).
errstat	Output	7	Error status field. Debug information delivered by the HMC device; see the HMC specification for more information.
dinv	Output	1	Data invalid; response data should be completely ignored if this is asserted.

How to Build a Project

The HMC controller distribution comes with pre-compiled projects for the evaluation systems, including projects for the example application. To create your own project targeting a Micron evaluation system, from the distribution, copy the appropriate project based on the board you are targeting. Replace the distributed sample code with your own HDL. Be sure to update your PicoDefines.v header file in your project, as described in the Pico Framework documentation. At the very least, you will likely need to update the name of your user module, but you may also need to update the number of streams.

Project Restrictions

When using the HMC controller in your project, the following restrictions/requirements apply:

- You must have the PicoBus in your design. The HMC controller uses this for its primary communication with the host to bring up the HMC link. In particular, the controller requires a 32-bit PicoBus.
- Users must share the PicoBus address space with the HMC controller. The HMC controller leaves addresses 0x10000 - 0xF00000 for user logic.
- READ and WRITE operations will not cross a row (page) boundary within the DRAM array (within the HMC). The HMC controller assumes the user does not break this rule.
- Addresses for all commands must be 16B aligned.

User Interface Commands

Table 3: Supported Commands

Verilog Define	Command Name
HMC_CMD_RD	READ
HMC_CMD_WR	WRITE
HMC_CMD_WR_NP	NON-POSTED WRITE
HMC_CMD_BW	BIT WRITE
HMC_CMD_BW_NP	NON-POSTED BIT WRITE
HMC_CMD_A16	ATOMIC 16-BYTE ADD IMMEDIATE
HMC_CMD_A16_NP	NON-POSTED ATOMIC 16-BYTE ADD IMMEDIATE
HMC_CMD_A8	DUAL 8-BYTE ADD IMMEDIATE
HMC_CMD_A8_NP	NON-POSTED DUAL 8-BYTE ADD IMMEDIATE

Note: 1. These user commands are defined within hmc_def.vh, which is a header file that is included in the HMC controller distribution.

READ Command

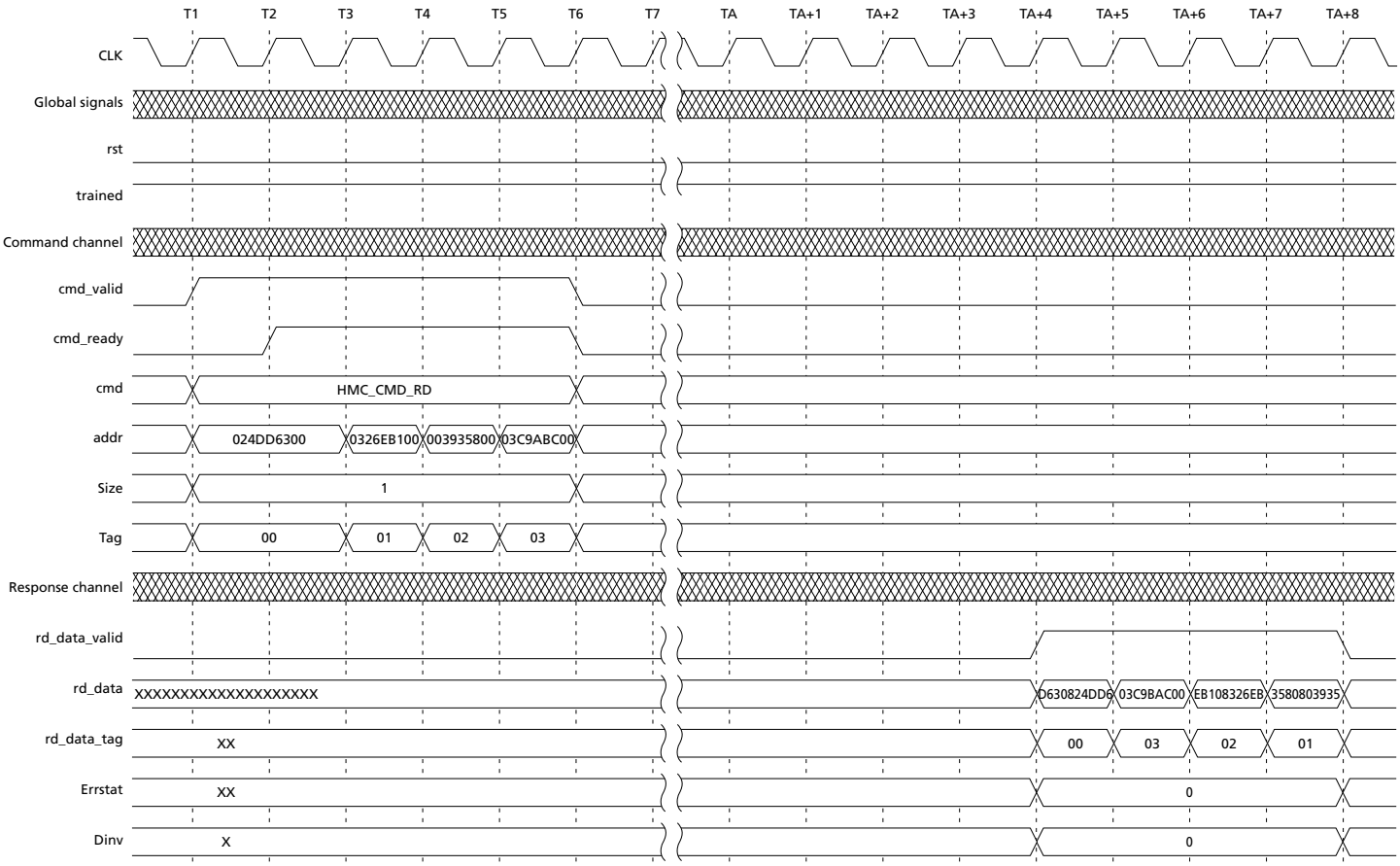
Read requests to the controller are simpler than write requests because they do not require write data. However, each read does produce exactly one read response. The HMC routinely reorders read requests to service them as efficiently as possible. Therefore, read responses may arrive at the read data port in a different order from which they were issued to the HMC. To resolve this, the controller includes a tag field that correlates read data with its originating READ command.

To begin a READ command, the read address (in bytes aligned to a 16-byte boundary) and the read size (in units of 16-byte words) are placed on the command port. The read address must be placed on the address port, the read size on the size port, with the command set to zero to perform a read.

A unique tag must be on the proper port. The tag must be unused by any other read request that has not returned a read response. The tag can be reused after a read response is returned.

When the user logic asserts the cmd_valid signal, the controller will assert the cmd_ready signal. The controller can also assert the cmd_ready signal later, which indicates readiness to accept a new read request.

Figure 3: Read and Response Timing Diagram



The figure above shows both a series of read requests to the HMC controller and the returned read data. The first request is made when both cmd_valid and cmd_ready are asserted for tag = 0. Subsequent requests are made with random addresses and incrementing tag numbers. The HMC controller may or may not assert cmd_ready in the same cycle that cmd_valid is asserted. The cmd_valid signal should not depend on cmd_ready.

The best read performance is achieved when many read requests are pipelined to the HMC controller. There is no need to wait for the first read response before sending a subsequent read request to the HMC controller. In the example shown in the figure above, four read requests are issued before any read data is returned. Each request attempts to read exactly 16 bytes of data from the HMC.

After a variable amount of latency that depends on the HMC load, the HMC controller returns the read data for the read requests. Because this read data may be returned out of order, users must pay close attention to the rd_data_tag. When the HMC controller has valid read data, it asserts the rd_data_valid signal. When this occurs, the user logic must be ready to accept the read data. Therefore, users must be careful not to issue read requests to the HMC unless they have enough buffer space in their own logic to store the read data.

The figure also shows the read response. In this example, 16 bytes at a time are read (size = 1) from random addresses from a previously initialized memory where the data stored at each 32-bit entry equals the data's address. In the example, the controller asserts the rd_data_valid signal as soon as the read data has been prepared for delivery to the user logic.

The HMC controller can receive response data for different read requests on consecutive clock cycles; a dead cycle between response data for different requests is not needed. When a read request size is larger (size > 1) the read data for the request is transferred in multiple rx_clk cycles. If responses always came back in the order that requests were issued, in our example we would expect to see tag = 3 returned last. However, the HMC reorders the read requests and returns the read data for tag = 3 second.

Although the HMC may reorder packets, we will never interleave the response data for two different packets.

WRITE Command

Because write requests to the HMC controller require write data, they are more complex than read requests.

To begin a WRITE command, first set the write address and write size on the command port. The write address should be specified in bytes, and the write size should be specified in terms of 16-byte words. The write address must be aligned to a 16-byte boundary.

Specifically, the write address is placed on the address port, the write size on the size port, and the cmd is set to HMC_CMD_WR. After the user logic asserts the cmd_valid signal, the HMC controller will assert the cmd_ready signal, zero or more cycles later when it is ready to accept the new write request.

In parallel, the user logic should prepare to deliver the write data to the HMC controller. The interface for the write data is essentially a stream interface containing data, a valid

signal and a ready signal. Write data should be delivered to the HMC controller in the identical order that WRITE commands are issued.

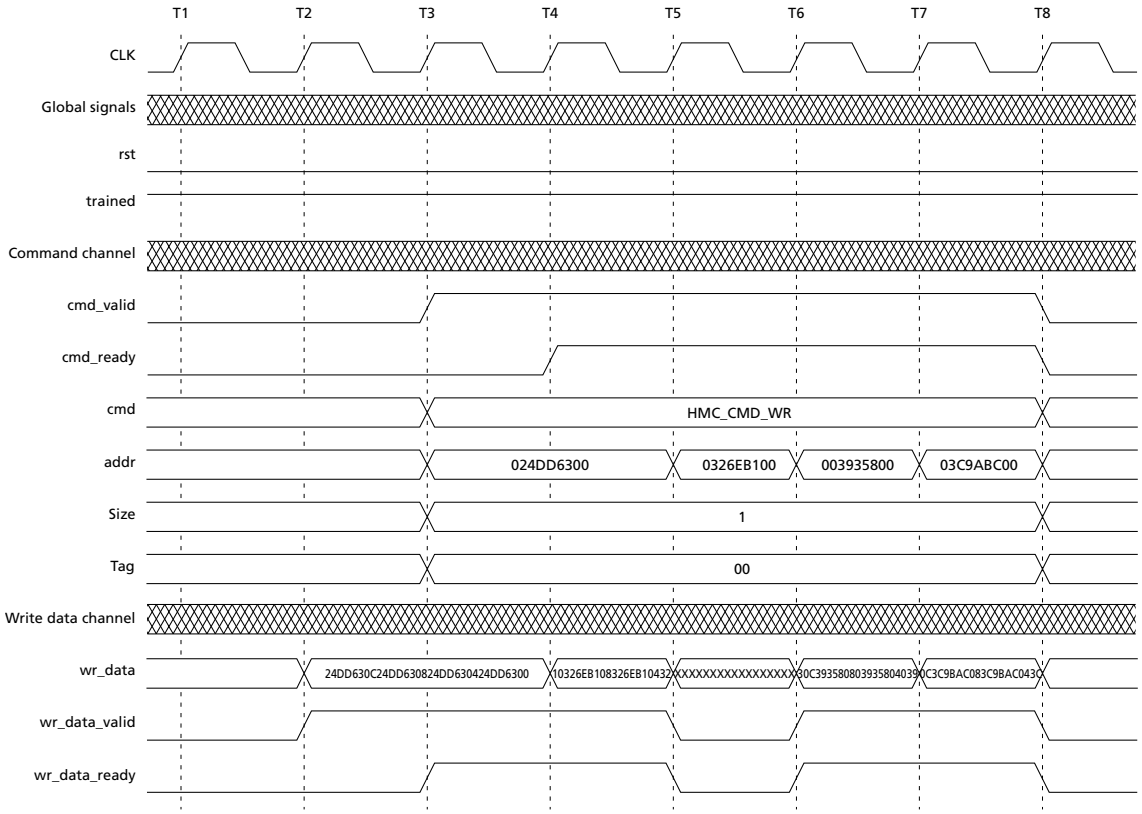
The write data interface is simple. First, users set the data on the wr_data bus and then assert wr_data_valid. The HMC controller will then assert the wr_data_ready signal, zero or more cycles later when it is ready to accept this new write data.

This write data interface should be independent of the command interface. That is, the wr_data_valid signal should not depend on the cmd_valid or cmd_ready_signals. Also, the cmd_valid signal should not depend on the wr_data_valid or wr_data_ready signals.

A posted WRITE command does not produce any response, so there is no need to match response data with the command. Also, in a posted write request, the HMC controller does not use the tag field, which enables the user to populate this field with any value.

Note that the write data can be delivered to the HMC controller either at the same time as the command (in the same clock cycle) or after the command has been accepted by the controller. There may be a small amount of buffering in the controller for write data, which would mean that the write data could be delivered a few cycles before the write command. However, we make no guarantees about the availability of this buffering, and users should be careful to avoid a deadlock situation that might arise if they introduce a dependency in their logic between the write data and the write command.

Figure 4: Writing Data to the HMC



The figure above is an example of writing random memory addresses with 32-bit pieces of data equal to their addresses. For example, when writing the address 0x0, we would write: 00000003_00000002_00000001_00000000.

As shown in the timing diagram above, wr_data_valid is asserted one cycle before asserting cmd_valid for the first time. The HMC controller accepts this write data, even one cycle before the first address is sent to the controller. Note that we could have asserted the wr_data_valid signal in the same cycle that we asserted the cmd_valid signal. These valid signals are independent.

After the write address is sent to the HMC controller and when cmd_valid and cmd_ready are asserted, the HMC controller performs the WRITE operation. In this non-posted write example, the tag never changes because there is no need to match response data to a unique tag.

On the command port, the cmd_valid signal is asserted one cycle before the HMC controller asserts the cmd_ready signal. Also, the cmd_ready signal is asserted for four consecutive cycles. And because cmd_valid is asserted at the same time as cmd_ready for four cycles, four write requests are issued.

Note that we could have de-asserted cmd_valid during T4 in our previous example. If we had, since cmd_valid and cmd_ready were not both asserted in the same cycle until T5, we would not have sent the first command to the HMC controller until T5.

If the write data port is prepared to send data, the wr_data_valid signal is asserted one cycle before the HMC controller asserts the wr_data_ready signal. However, if the logic has on-hand only the first two pieces of data to write to the HMC, two pieces of data are sent to the HMC controller. After a one-cycle pause, the wr_data_valid signal is asserted, and the HMC controller is immediately ready to accept that data; that is, the wr_data_ready signal has been asserted in the same cycle.

The timing relationship between the WRITE command and the write data demonstrates that write data may be sent to the HMC controller previous to or concurrent with the WRITE command. It is also valid to send the write data to the HMC controller after sending the WRITE command.

NON-POSTED WRITE Command

Non-posted writes are very similar to posted writes, except that they return a response. This response tells the user when the write has completed by returning the tag for the original write request. If errors occurred during the transfer or execution of the WRITE request command, status will be included in the ERRSTAT field of the write response packet. Non-posted writes can be very useful for determining when it is safe to read a memory location that was written, for example, avoiding read-after-write hazards in an instruction stream.

To begin a NON-POSTED WRITE command, first set the write address and write size on the command port. The write address should be specified in bytes, and the write size should be specified in terms of 16-byte words. The write address must be aligned to a 16-byte boundary.

Specifically, the write address is placed on the address port, the write size on the size port, and the cmd is set to 1. After the user logic asserts the cmd_valid signal, the HMC controller will assert the cmd_ready signal, zero or more cycles later when it is ready to accept the new write request.

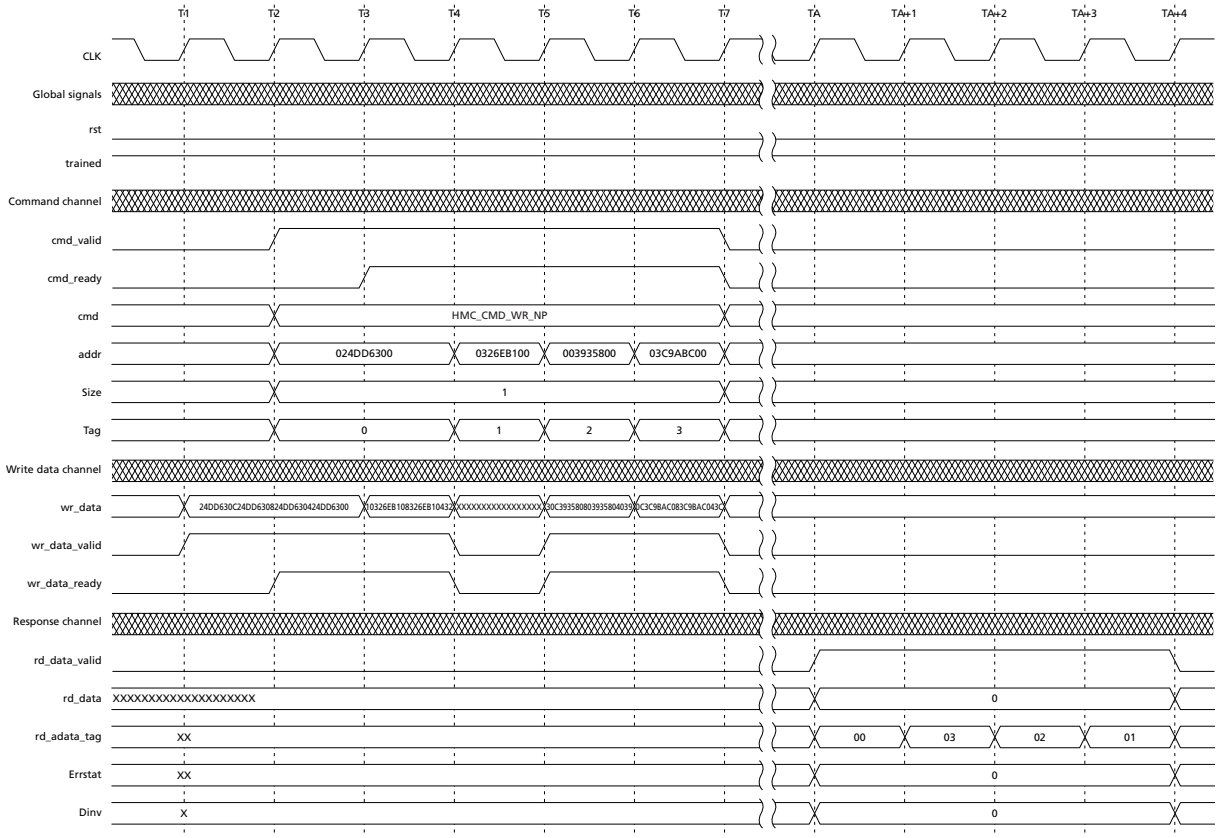
In parallel, the user logic should prepare to deliver the write data to the HMC controller. The interface for the write data is essentially a stream interface containing data, a valid signal, and a ready signal. Write data should be delivered to the HMC controller in the identical order that WRITE commands are issued.

The write data interface is simple. First, users set the data on the wr_data bus and then assert wr_data_valid. The HMC controller will then assert the wr_data_ready signal, zero or more cycles later when it is ready to accept this new write data.

This write data interface should be independent of the command interface. That is, the wr_data_valid signal should not depend on the cmd_valid or cmd_ready_signals. Also, the cmd_valid signal should not depend on the wr_data_valid or wr_data_ready signals.

After a variable amount of latency, the HMC controller returns a response, which signals that the write has completed. It does this by setting the rd_data_tag field to the tag of the original write request and asserting the rd_data_valid signal. Note that this will look very similar to a response from a READ command with size = 1, so it is up to the user to know that this is a response for a WRITE command and not for a read.

Figure 5: Writing Data to the HMC



The figure above is an example of writing random memory addresses with 32-bit pieces of data equal to their addresses. For example, when writing the address 0x0, we would write: 0x0000000300000002000000001_00000000.

As shown in the timing diagram above, wr_data_valid is asserted one cycle before asserting cmd_valid for the first time. The HMC controller accepts this write data, even one cycle before the first address is sent to the controller.

After the write address is sent to the HMC controller and when cmd_valid and cmd_ready are asserted, the HMC controller performs the WRITE operation. Because this is a posted-write, and because WRITE commands may be executed out-of-order by the HMC, we must use a unique tag for each WRITE command.

On the command port, the cmd_valid signal is asserted one cycle before the HMC controller asserts the cmd_ready signal. Also, the cmd_ready signal is asserted for four consecutive cycles. And because cmd_valid is asserted at the same time as cmd_ready for four cycles, four write requests are issued.

If the write data port is prepared to send data, the wr_data_valid signal is asserted one cycle before the HMC controller asserts the wr_data_ready signal. However, if the logic has on-hand only the first two pieces of data to write to the HMC, two pieces of data are sent to the memory. After a one-cycle pause, the wr_data_valid signal is asserted, and the HMC controller is immediately ready to accept that data; that is, the wr_data_ready signal has been asserted in the same cycle.

The timing relationship between the WRITE command and the write data demonstrates that write data may be sent to the HMC controller previous to or concurrent with the WRITE command. It is also valid to send the write data to the HMC controller after sending the WRITE command.

The figure also shows the write response. In the example, the controller asserts the rd_data_valid signal when the write has completed in the HMC. Note that we use the tag to label this response as a write (since the command with that matching tag was a non-posted write), and therefore the corresponding read data should be ignored. The only pertinent fields for a write response are the errstat and dinv fields, which in this case are all zero, meaning that our write completed successfully. In the case that they are non-zero, refer to the HMC specification to decode the errstat field.

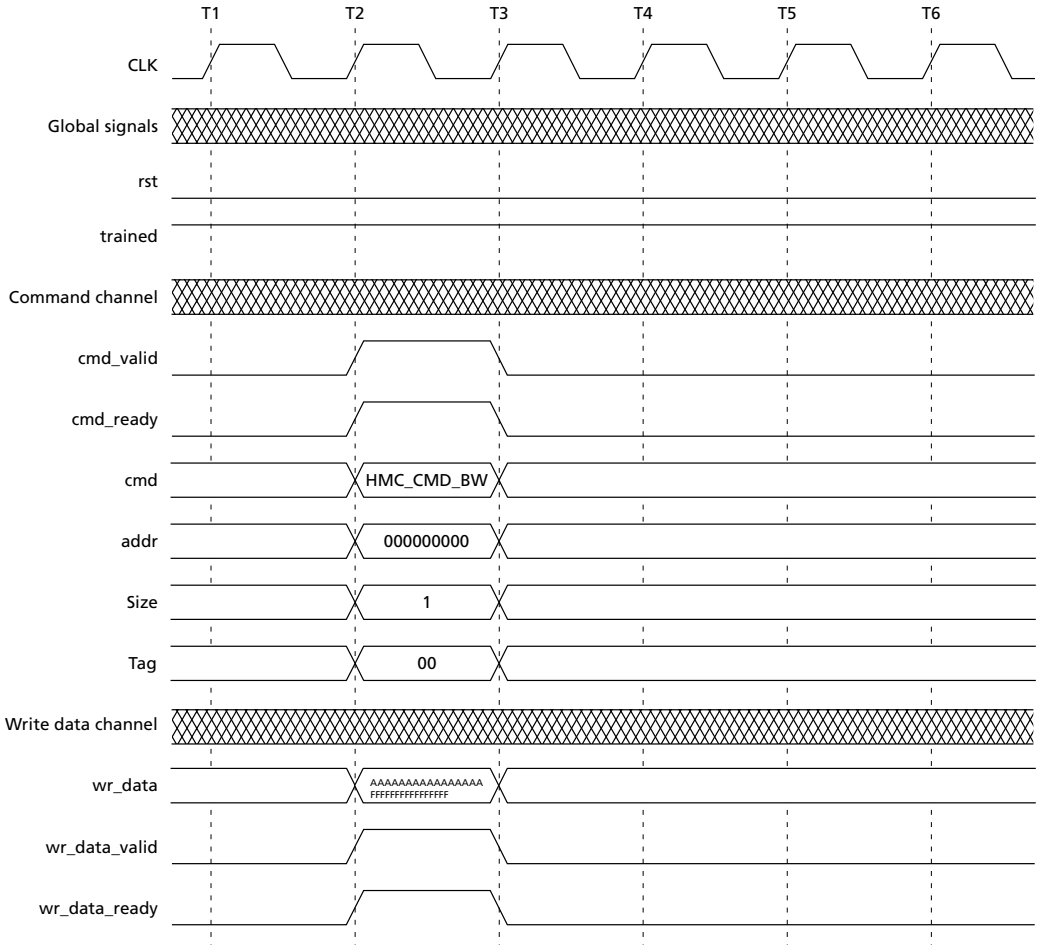
Note that all write responses will be exactly one cycle. Similar to the read, the HMC may reorder operations within the cube. In this case, the non-posted write with tag = 3 must have been executed by the HMC first, because the response was returned by the HMC first.

BIT WRITE Command

Assuming that each element of an array stored in HMC is 4 bytes, the HMC controller BIT WRITE command enables users to write a new value to only one element of the array. The BIT WRITE command enables users to write from 0 to 8 bytes to the HMC with single-bit granularity.

The posted bit write is performed in a similar manner to that of a 16-byte posted write, with two small differences. First, the cmd must be set to 0xA. Second, the format of the data on the wr_data bus is slightly different. The data to be written to the HMC should be placed in the LS 8 bytes of the wr_data bus. The MS 8 bytes of the wr_data bus serve as a bit inhibitor that is applied to the data in the LS 8 bytes. To clarify, bit 64 is a mask for the data in bit 0, bit 65 is a mask for the data in bit 1, and so on. Only those data bits (in the LS 8 bytes of wr_data) whose corresponding mask bit is 0 will actually be updated in the HMC memory. In other words, if a mask bit is 1, the corresponding data bit in the HMC memory will not be updated, but will remain at its current value.

Figure 6: Posted Bit Write Timing Diagram



The figure above shows a bit WRITE operation that updates to a 1 every other bit of the data at address 0. To do this, set the cmd and pass it to the HMC controller. Next, set up the data to be written. In this case, the update operation sets bits to 1, so set the LS 64 bits of the wr_data to all 1s. Since we only want to update every other bit, set every other bit of the MS 64 bits of wr_data to 0. With the mask bits set as 0xAAAAAAAA, only the even-numbered bits will be updated; the odd-numbered bits will retain their previous values.

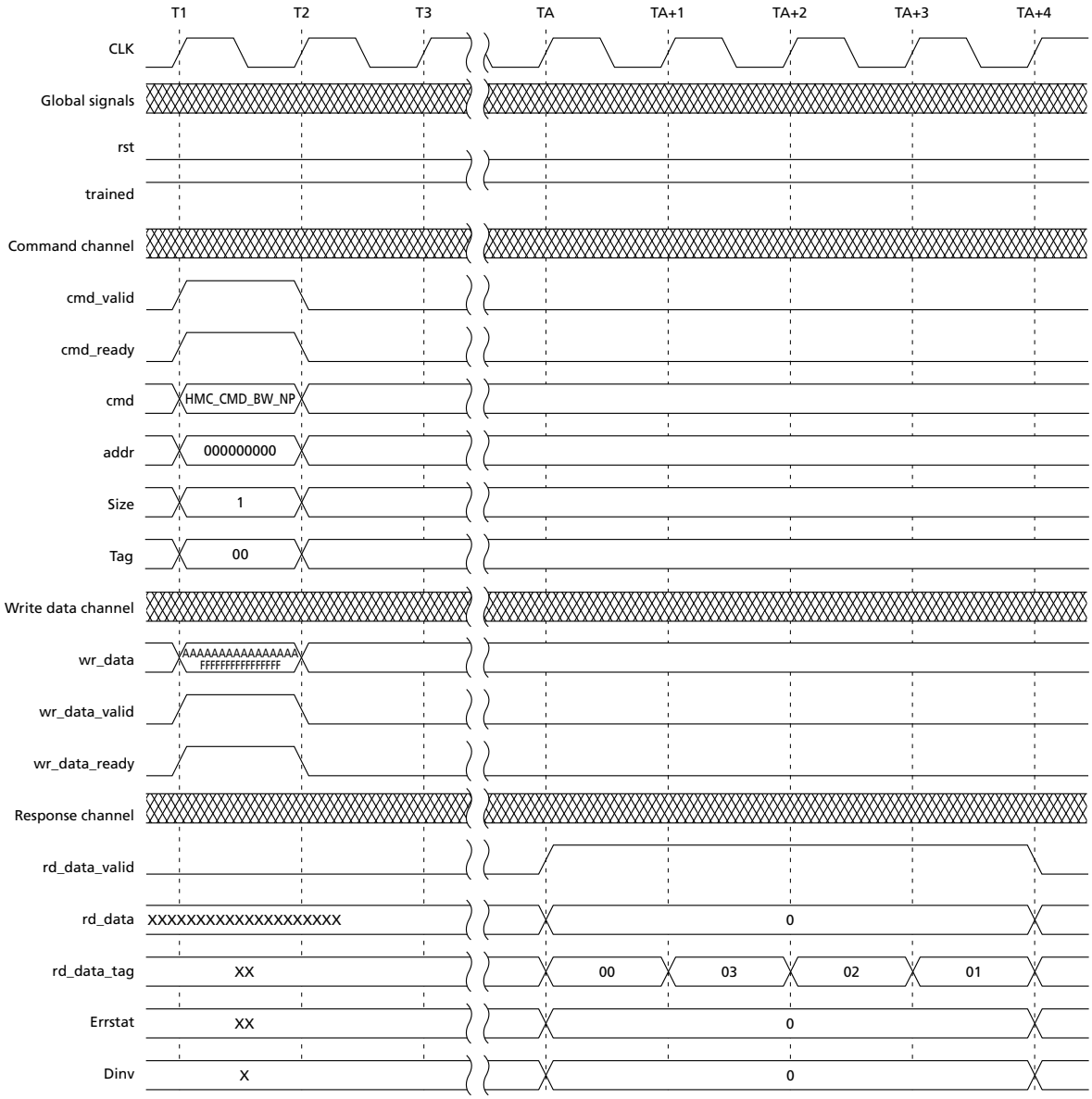
NON-POSTED BIT WRITE Command

The BIT WRITE command enables writing 0 to 8 bytes to the HMC with single-bit granularity. In the case of the NON-POSTED BIT WRITE, we get a response indicating the status of the write.

The NON-POSTED BIT WRITE is similar to a 16-byte NON-POSTED WRITE with two differences: First, the CMD must be set to 0x2 and data on the wr_data bus is formatted slightly different. And second, data written to the HMC is placed in the LS 8 bytes of the wr_data bus.

The MS 8 bytes of the wr_data bus serve as a bit inhibitor applied to the data in the LS 8 bytes. Bit 64 masks data in bit 0, bit 65 masks data in bit 1, and so on. In the LS 8 bytes of wr_data, each data bit with a corresponding mask bit 0 in the MS 8 bytes of the wr_data is updated in HMC memory. However, if a mask bit is 1, its corresponding data bit in the HMC memory is not updated, but remains at its current value.

Figure 7: Non-Posted Bit Write Timing Diagram



The figure above shows a NON-POSTED BIT WRITE operation that updates every other bit of data at addresses 0 to 1. First, the CMD is set and its value passed to the HMC controller. Next, the data to be written is set up: The update operation sets all LS 64 bits of the WR_DATA to 1s.

Every other bit of the MS 64 bits of WR_DATA is set to zero so that only every other bit of LS 64 bits of the WR_DATA will be updated. Therefore, with the mask bits set as 0xAAAAAAAA, only the even-numbered bits will be updated. The odd-numbered bits will retain their previous values.

The difference in this example versus the POSTED BIT WRITE example is that now we get a response. The timing diagram clearly shows a response for our BIT WRITE command being returned by the HMC controller after a variable latency. Note that since this is a write response, the read data should be treated as a "Don't Care." The only pertinent fields are the errstat and dinv, which in this case are all zeroes. In the case that they are non-zero, refer to the HMC specification to decode the errstat field.

ATOMIC 16-BYTE ADD IMMEDIATE Command

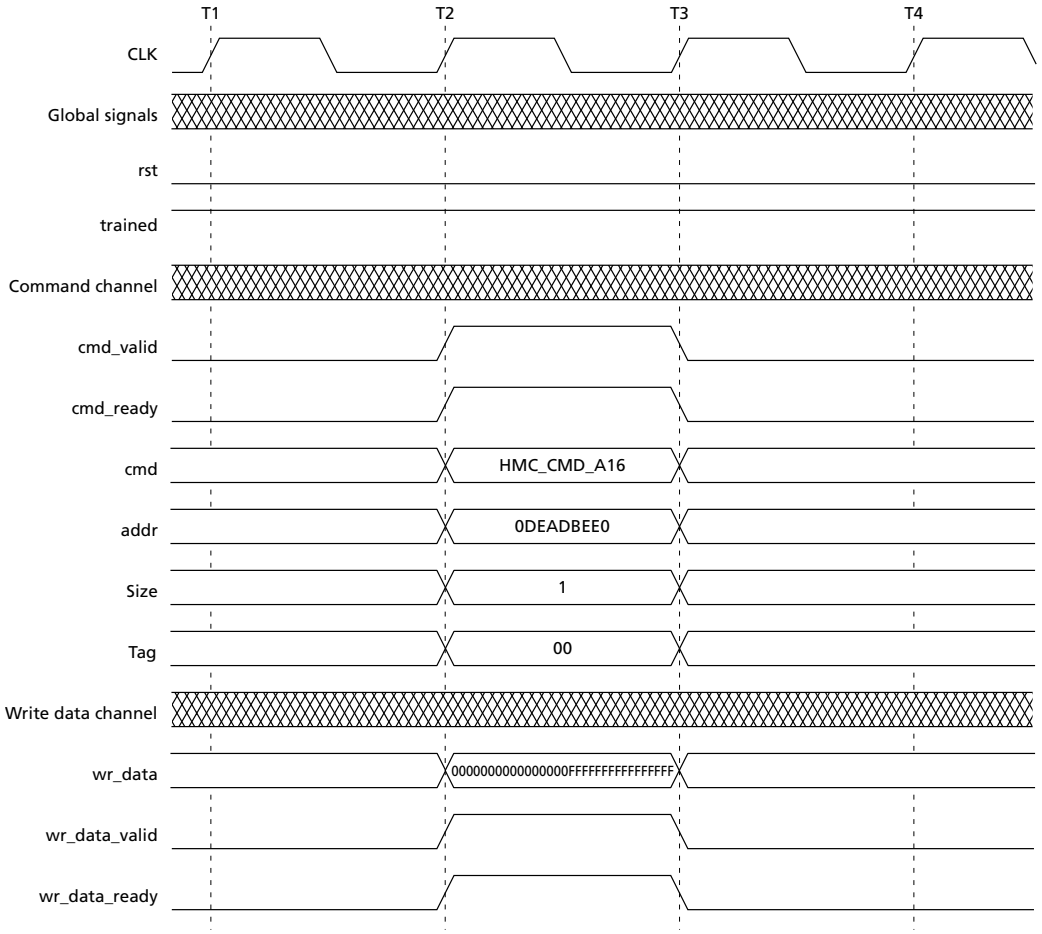
The ADD IMMEDIATE command allows users to atomically update a 16B entry in the memory with an 8B 2's complement signed integer. Atomic requests involve reading 16 bytes of data from the HMC memory, performing an operation on the data through the use of a 8-byte operand, and then writing the results back to the same location in HMC memory. The read-update-write sequence occurs atomically, meaning that no other request can access the same HMC memory bank until the write of the atomic request is complete. This section describes the proper method to execute the ADD IMMEDIATE command using the HMC controller.

For simplicity, assume the data is already in the memory. Like standard reads and writes, the address must be aligned to a 16B boundary. Operating upon these assumptions and constraints, there are only a couple differences for the ADD IMMEDIATE command, as compared to the WRITE command. First, since we are going to do a read-modify-write on 16B of data, our size must be exactly 0x1. Second, the MS 8B of the write_data should be set to all 0s. Finally, the LS 8B of the write_data should be the 2's complement representation of the integer that we want to add to the data that currently resides at our address in memory. The following table shows the breakdown of data that should be placed on the write_data bus.

Table 4: Write_Data Bus

Bytes	15:8	7:0
Field	0x0000000000000000	2's complement signed integer

Figure 8: Posted Atomic 16-Byte Add Immediate Timing Diagram



The figure above shows a simple example where we are decrementing the value stored at address 0xDEADBEE0 by 1. In this example, we first set our addr to 0xDEADBEE0, the cmd to 0xC, and assert the cmd_valid signal until cmd_ready is asserted by the HMC controller. In parallel, we set the MS 8B of write_data to 0 and the LS 8B to 0xFFFFFFFFFFFFFFFF (for example, –1). We then assert the write_data_valid signal and wait for the HMC controller to assert the write_data_ready signal.

> **Note:** If the result exceeds 2^{127}, the carry out of the most significant bit is dropped and the result rolls over.

NON-POSTED 16-BYTE ADD IMMEDIATE Command

The NON-POSTED ADD IMMEDIATE command enables users to atomically update a 16-byte entry in the memory with an 8 bytes 2's complement signed integer. Atomic requests read 16 bytes of data from the HMC memory and perform an operation on the data through a 8-byte operand and then write the results back to the same location in HMC memory. The read-update-write sequence occurs atomically, meaning that no other request can access the same HMC memory bank until the write of the atomic request is complete. This section describes the proper method to execute the ADD IMMEDIATE command using the HMC controller.

Assuming that the data is already in the memory, such as a standard read and write, the address must be aligned to a 16-byte boundary. Compared to the WRITE command, there are only three differences for the ADD IMMEDIATE command.

First, for a READ-MODIFY-WRITE command on 16 bytes of data, size must be exactly 0x1.

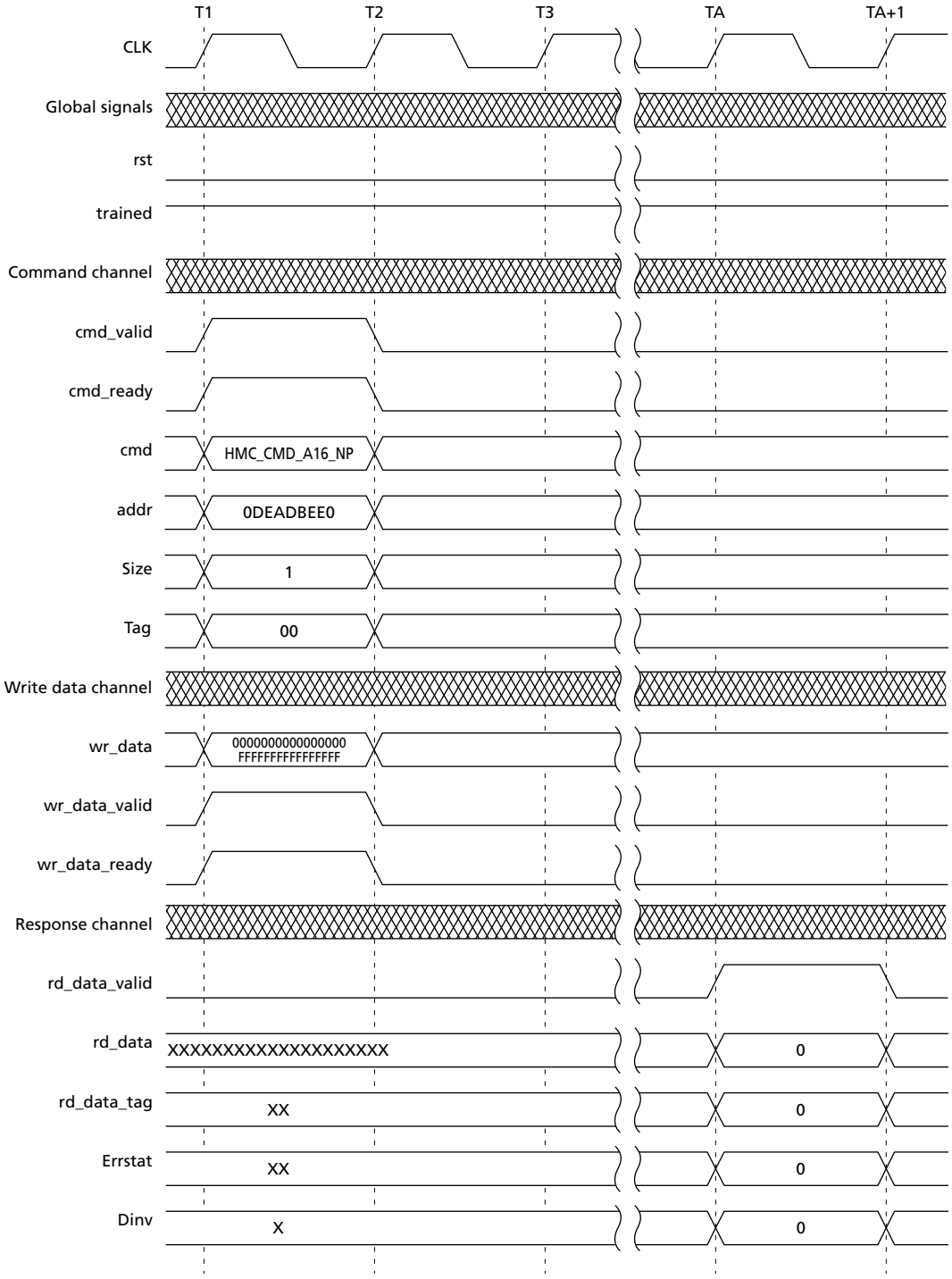
Second, the MS 8 bytes of the write_data should be set to zero.

Finally, the LS 8 bytes of the write_data should be the 2's complement representation of the integer added to the data at the memory address. The following table shows the breakdown of data that should be placed on the write_data bus.

Table 5: Write_Data Bus

Bytes	15:8	7:0
Field	0x0000000000000000	2's complement signed integer

Figure 9: Non-Posted Atomic 16-Byte Add Immediate Timing Diagram



The figure above shows an example where the value stored at address 0xDEADBEE0 is decremented by one. In this example, ADDR is set to 0xDEADBEE0, and CMD is set to 0x4. The CMD_VALID signal is asserted until the HMC controller asserts the cmd_ready signal is asserted by the HMC controller.

In parallel, the MS 8 bytes of WRITE_ DATA are set to zero and the LS 8 bytes are set to 0xFFFFFFFFFFFFFFFF (–1). The write_data_valid signal is asserted and then the HMC controller asserts the write_data_ready signal.

> **Note:** If the result exceeds 2^{127}, the carry out of the most significant bit is dropped and the result rolls over.

The difference in this example versus the ATOMIC 16-BYTE ADD IMMEDIATE example is that now we get a response. The timing diagram clearly shows a response for our ADD IMMEDIATE command being returned by the HMC controller after a variable latency. Note that since this is a write response, the read data should be treated as a "Don't Care." The only pertinent fields are the errstat and dinv, which in this case are all zeroes. In the case that they are non-zero, refer to the HMC specification to decode the errstat field.

DUAL 8-BYTE ADD IMMEDIATE Command

The dual ADD IMMEDIATE command enables users to atomically update two consecutive 8-byte entries in the memory, where each update uses its own 4-byte 2's complement signed integer. The read-update-write sequence occurs atomically, meaning that no other request can access the same HMC memory bank until the write of the atomic request is complete. The following is an explanation of the proper way to execute that command using the HMC controller.

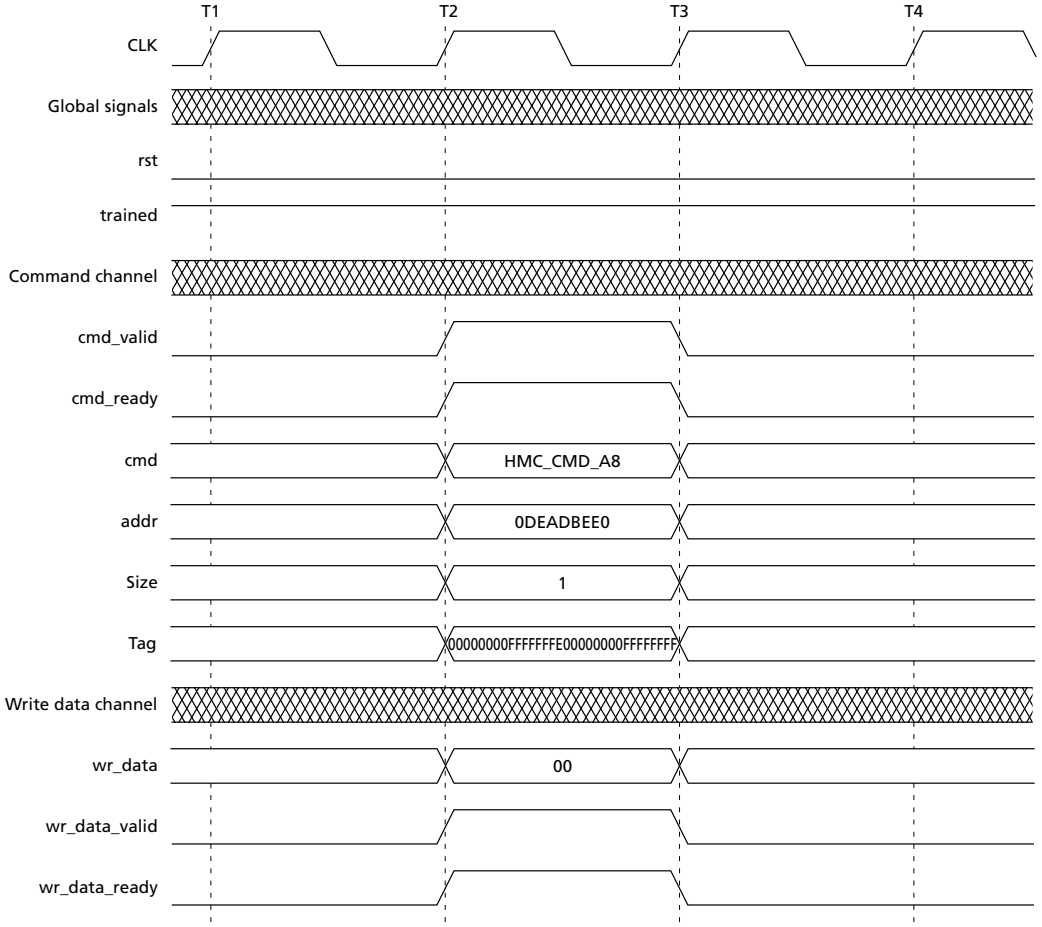
Assume that there are two 8-byte pieces of data to update where the data resides at consecutive addresses. As with standard READ and WRITE commands, the address must be aligned to a 16-byte boundary. Compared to the WRITE command, the dual 8-byte and single 16-byte ADD IMMEDIATE commands must be exactly 0x1 in size. Also, write_data bytes 15 to 12 and 7 to 4 of the write_data should all be set 0s. The remaining bytes (11 to 8 and 3 to 0) make up the two 4-byte 2's complement signed integers.

The integer in bytes 11 to 8 of write_data are added to the MS 64 bits of the data in the HMC memory. Similarly, the integer in bytes 3 to 0 of write_data are added to the LS 64 bits of the data in the HMC memory. The following table shows breakdown of data placed on the write_data bus.

Table 6: Write_Data Bus

Bytes	15:12	11:8	7:4	3:0
Field	0x00000000	2's complement signed integer	0x00000000	2's complement signed integer

Figure 10: Posted Atomic Dual 8-Byte Add Immediate Timing Diagram



The figure above shows an example of updating two consecutive values at address 0xDEADBEE0. The first value is decremented by 1, and the second value is decremented by 2. ADDR is set to 0xDEADBEE0, and cmd is set to 0xE. The cmd_valid signal is asserted until the HMC controller asserts the cmd_ready signal.

In parallel, bytes 11 to 8 of write_data are set to 0xFFFFFFFE (–2); bytes 3 to 0 are set to 0xFFFFFFFF (–1). The write_data_valid signal is asserted, and then HMC controller asserts the write_data_ready signal.

> **Note:** If the result exceeds 2^{63}, the carry out of the most significant bit is dropped and the result rolls over.

NON-POSTED ATOMIC DUAL 8-BYTE ADD IMMEDIATE Command

The dual ADD IMMEDIATE command enables users to atomically update two consecutive 8-byte entries in the memory, where each update uses its own 4 byte 2's complement signed integer. The read-update-write sequence occurs atomically, meaning that no other request can access the same HMC memory bank until the write of the atomic request is complete. The following is an explanation of the proper way to execute that command using the HMC controller.

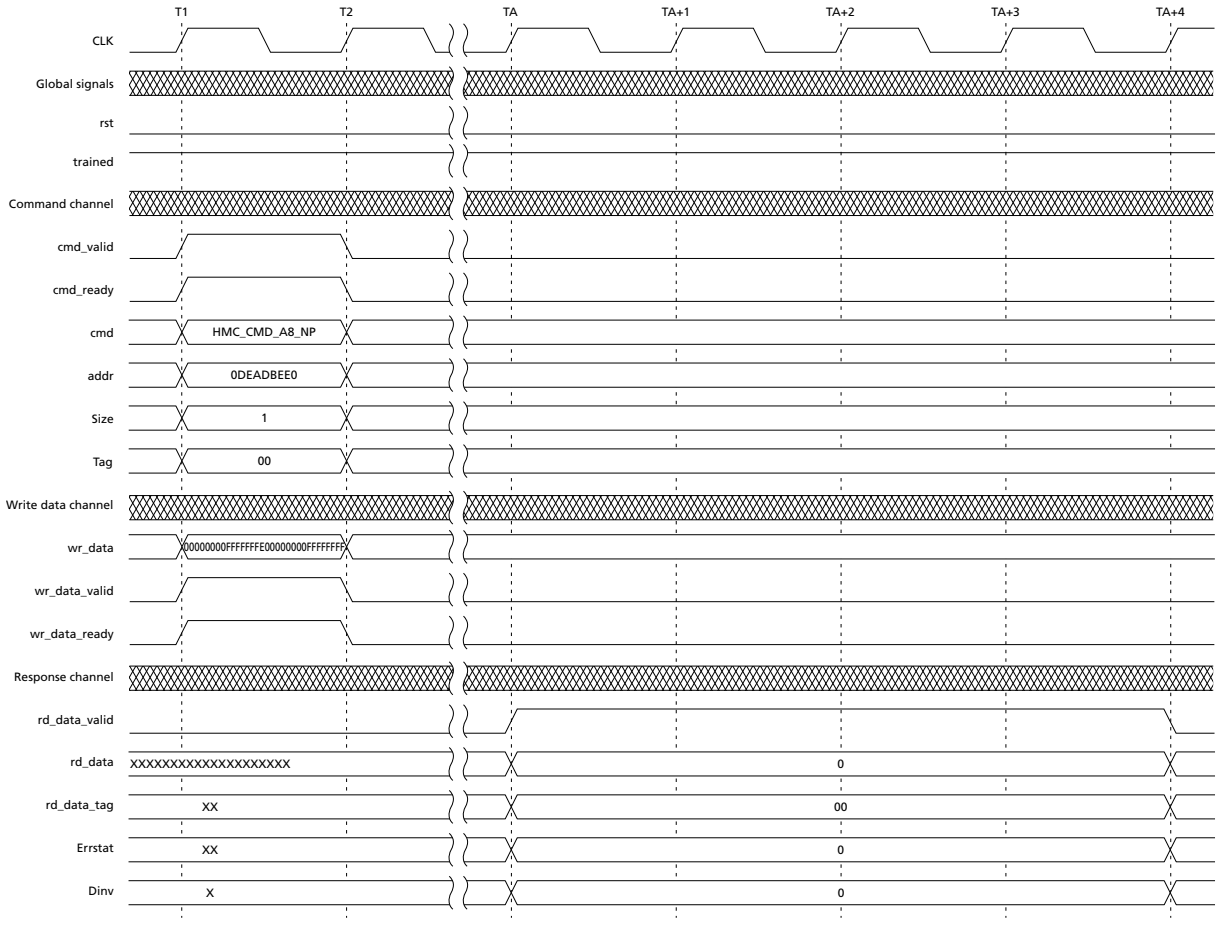
Assume that there are two 8-byte pieces of data to update where the data resides at consecutive addresses. As with standard READ and WRITE commands, the address must be aligned to a 16-byte boundary. Compared to the WRITE command, the dual 8-byte and single 16-byte ADD IMMEDIATE commands must be exactly 0x1 in size. Also, write_data bytes 15 to 12 and 7 to 4 of the write_data should all be set 0s. The remaining bytes (11 to 8 and 3 to 0) make up the two 4-byte 2's complement signed integers.

The integer in bytes 11 to 8 of write_data are added to the MS 64 bits of the data in the HMC memory. Similarly, the integer in bytes 3 to 0 of write_data are added to the LS 64 bits of the data in the HMC memory. The following table shows breakdown of data placed on the write_data bus.

Table 7: Write_Data Bus

Bytes	15:12	11:8	7:4	3:0
Field	0x00000000	2's complement signed integer	0x00000000	2's complement signed integer

Figure 11: Non-Posted Atomic Dual 8-Byte Add Immediate Timing Diagram



The figure above shows an example of updating two consecutive values at address 0xDEADBEE0. The first value is decremented by 1, and the second value is decremented

by 2. ADDR is set to 0xDEADBEE0, and cmd is set to 0xE. The cmd_valid signal is asserted until the HMC controller asserts the cmd_ready signal.

In parallel, bytes 11 to 8 of write_data are set to 0xFFFFFFFE (–2); bytes 3 to 0 are set to 0xFFFFFFFF (–1). The write_data_valid signal is asserted, and then HMC controller asserts the write_data_ready signal.

> **Note:** If the result exceeds 2^{63}, the carry out of the most significant bit is dropped and the result rolls over.

The difference in this example versus the DUAL 8-BYTE ADD IMMEDIATE example is that now we get a response. The timing diagram clearly shows a response for our ADD IMMEDIATE command being returned by the HMC controller after a variable latency. Note that since this is a write response, the read data should be treated as a don't-care. The only pertinent fields are the errstat and dinv, which in this case are all zeroes. In the case that they are non-zero, refer to the HMC specification to decode the errstat field.

Bringup Sequence

The HMC controller and the memory must be initialized before it can be used. The initialization sequence instructs the controller to train its link with the HMC and initialize important registers. Initialization is automatically performed by the Pico Computing Framework the first time that ReadRam or WriteRam are called in software. During execution of the bringup sequence, any commands sent to the controller from the user HDL will not be acknowledged.

To communicate with the controller, the Pico Computing device driver must be installed in the host system. After the bringup sequence completes, the user RTL can send commands to the controller.

Evaluation Systems

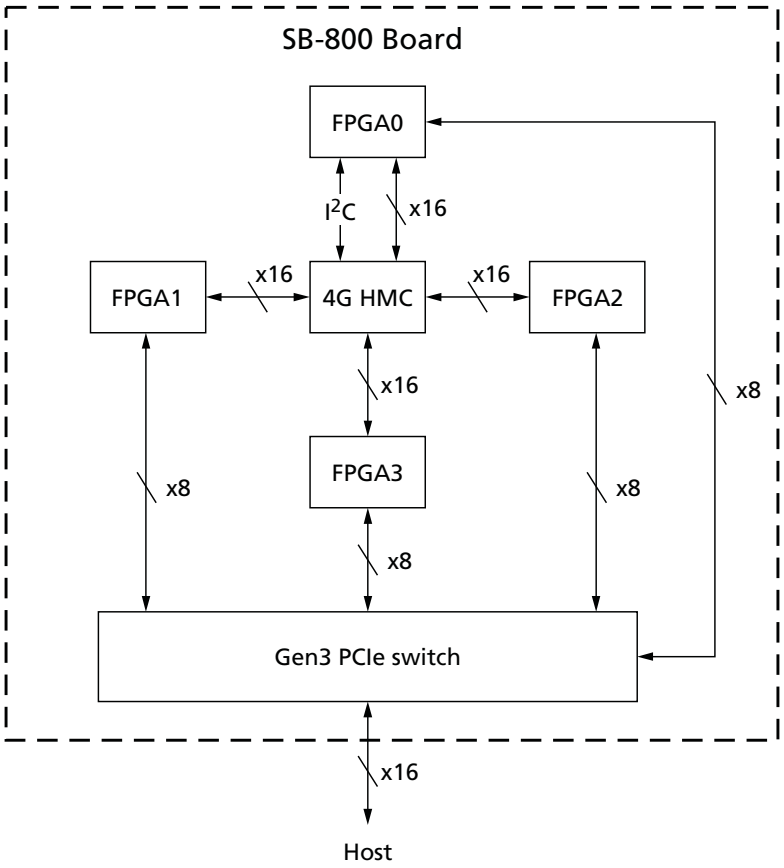
The table below and the two figures that follow, SB-800 – 4 x Full-Width 10G HMC Links and AC-510– 2 x Half-Width 15G HMC Links show the two types of available PCIe-based FPGA HPC boards to enable a fast in-system evaluation of the HMC controller. These boards insert into a standard Intel PC platform with double-wide Gen3 PCIe slots.

The boards are driven via the Pico Computing Framework composed of a PCIe bus controller, FPGA bitfile downloads, a DMA engine, APIs, and so on. The Framework makes it easy to link host application software to the HMC while greatly simplifying HMC integration and usage. The resulting system runs under Ubuntu 14.04 LTS (64-bit), Pico Computing Framework 5.4.1.0, and HMC IP Release 4.0.

Table 8: Evaluation System – Available PCIe-Based FPGA HPC Boards

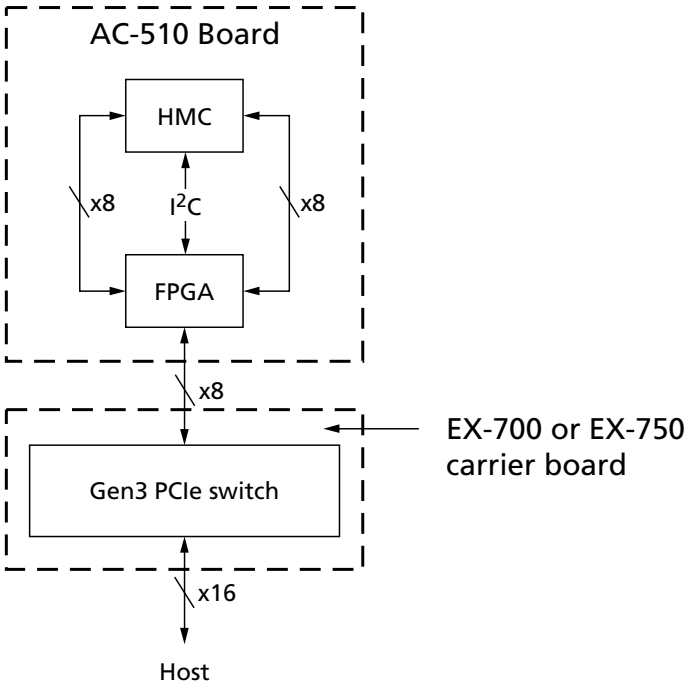
Board	Micron HMC Details	FPGA	HMC LInks
SB-800	1 x Micron 4GB HMC	4 x Altera 5SGXA9 FPGA	4 x full-width 10G HMC links
AC-510		1 x Xilinx XCKU060 FPGA	2 x half-width 15G HMC links

Figure 12: SB-800 – 4 x Full-Width 10G HMC Links



Note: 1. HMC x16 signals are between the HMC and FPGAs; PCIe x8 signals are between the PCIe switch and FPGAs.

Figure 13: AC-510 – 2 x Half-Width 15G HMC Links



Note: 1. HMC x8 signals are between the HMC and FPGA; the PCIe x8 signal is between the PCIe switch and FPGA.

Example Application

While most traditional memories are very good at reading and writing large sections of memory, such as kilobits, their performance suffers when accessing small, random locations. The HPCC RandomAccess benchmark is meant to measure a memory system's ability to perform these small, random accesses, which appear in numerous classes of algorithms and applications. Due to the parallelism within the HMC's logic layer, this type of application is a perfect demonstration of the HMC and HMC controller performance working in unison. In fact, the controller was specifically developed with this application in mind; it maintains a high streaming bandwidth for large reads and writes, but also provides the flexibility to deliver high performance when doing smaller accesses.

More specifically, the HPCC RandomAccess benchmark requires updates (read-modify-write) to randomly selected cells within a large array. The performance of systems running this benchmark is usually quoted in terms of Giga-Updates Per Second, thus the term GUPS.

Running the GUPS Benchmark

The GUPS benchmark is run using a pre-compiled bitfile, which is included in the HMC controller distribution. The only bitfiles included in the distribution are for the evaluation platforms, as listed earlier in the Evaluation System – Available PCIe-Based FPGA HPC Boards table.

The HMC controller leverages the Pico Computing Framework for low-level communication between the host and the FPGA. Therefore users must first install the appropriate Pico Computing package that is publicly available on the Pico Computing website. You can follow the Pico Computing documentation for installing this package.

After installing the Pico Computing package on your host system, you also need to install the HMC source in the correct location. We require that the HMC source be installed next to the Pico Computing package, which is likely located at /usr/src/picocomputing-X.X.X.X/, where X.X.X.X is a version number. After unzipping the HMC release .tgz file, copy the entire directory so it sits next to the Pico Computing package. Typically, the HMC source should be copied to /usr/src/HMC/ using the following steps:

1. *cd <dir containing hmc controller release>*
2. *tar -zxvf <hmc controller release>.tgz*
3. *sudo cp -r <unzipped hmc release dir name> /usr/src/*

At this point, you should expect to see the following folders in your /usr/src directory:

1. picocomputing-X.X.X.X.X
2. HMC

Now, you are ready to run the GUPS sample. To do this, first copy the GUPS sample directory from the Pico Computing package into your home directory. Then, cd into the GUPS/software directory and build the software using "make." You can then run the newly created executable. If you run the gups executable without any command-line arguments, it will tell you the arguments that it expects to see. Be sure to read the GUPS sample documentation, which is included in the Pico Computing package, for more information. These steps are outlined below:

1. *cp -r $PICOBASE/samples/GUPS $HOME*
2. *cd $HOME/GUPS/software*
3. *make*
4. *./gups*

The distributed application is extremely flexible, allowing users to control the parameters for running GUPS. See the table below.

Table 9: GUPS Parameters

Name	Option	Default	Description
Mode of operation	--rw, --ro, --wo	rw	Read-modify-write mode, Read-only mode, Write-only mode. Write-only mode ignores the *T* parameter, so it issues writes as fast as allowed by the HMC controller.
Number of user ports	--ports<P>	5	Enables the application to use exactly *P* user ports when communicating with the controller. More user ports means more parallel read and write requests. Range(*P*) = [1:5].

Table 9: GUPS Parameters (Continued)

Name	Option	Default	Description
Number of outstanding reads	--tags<T>	64	Enables the application to allow up to *T* read requests to be in flight at any given time per user port. More outstanding reads help hide the read latency and improve performance of the controller. Range(*T*) = [1:64].
Request size	--size<S>	1	Size of all read and write requests (in 16B words); for example, size = 1 means all read and write requests will be 16B. Range(*S*) = [1:8].

The most basic way to run the GUPS application is to just pass a bitfile. Note that the HMC release includes pre-built bitfiles for all the evaluation systems. Therefore, to run the GUPS sample, just call:

./gups --b <path to bitfile for your system>

This will run GUPS with all the default settings. Below is example output for running GUPS on the SB800:

micron@micron-ubuntu:~/GUPS/software$./gups --b ~/EX800_A9_GUPS.rbf

Loading FPGA with '/home/micron/EX800_A9_GUPS.rbf' ...

Loading FPGA with '/home/micron/EX800_A9_GUPS.rbf' ...

Loading FPGA with '/home/micron/EX800_A9_GUPS.rbf' ...

Loading FPGA with '/home/micron/EX800_A9_GUPS.rbf' ...

Calling ReadRam to bring up memory 1

Stopping GUPS modules on FPGA 0

Stopping GUPS modules on FPGA 1

Stopping GUPS modules on FPGA 2

Stopping GUPS modules on FPGA 3

Writing FPGA[0] user module reset = 0xF

Writing FPGA[1] user module reset = 0xF

Writing FPGA[2] user module reset = 0xF

Writing FPGA[3] user module reset = 0xF

gups_setup parameters:

mode: read-modify-write# modules: 4

 # modules: 4

packet size (in 16B flits): 1

 # tags: 0x40

linear mode : false

Setting up the GUPS modules on FPGA[0]

bringing user modules out of reset

CHECKING MAGIC #: 0x47555053

CHECKING VERSION #: 0x20000

Setting up the GUPS modules on FPGA[1]

bringing user modules out of reset

CHECKING MAGIC #: 0x47555053

CHECKING VERSION #: 0x20000

Setting up the GUPS modules on FPGA[2]

bringing user modules out of reset

CHECKING MAGIC #: 0x47555053

CHECKING VERSION #: 0x20000

Setting up the GUPS modules on FPGA[3]

bringing user modules out of reset

CHECKING MAGIC #: 0x47555053

CHECKING VERSION #: 0x20000

Module 1 GUPS: 0.037584

Module 2 GUPS: 0.037585

Module 3 GUPS: 0.037584

Module 4 GUPS: 0.037584

FPGA[0] Aggregate GUPS: 0.150337

Module 1 GUPS: 0.037584

Module 2 GUPS: 0.037584

Module 3 GUPS: 0.037584

Module 4 GUPS: 0.037584

FPGA[1] Aggregate GUPS: 0.150336

Module 1 GUPS: 0.037586

Module 2 GUPS: 0.037586

Module 3 GUPS: 0.037585

Module 4 GUPS: 0.037585

FPGA[2] Aggregate GUPS: 0.150342

Module 1 GUPS: 0.037586

Module 2 GUPS: 0.037586

Module 3 GUPS: 0.037585

Module 4 GUPS: 0.037585

FPGA[3] Aggregate GUPS: 0.150341

All-links-aggregate GUPS: 0.601356

This shows the speed of each module running on each FPGA. In the list above, four modules are running on four FPGAs. The last line shows the aggregate number of reads and/or writes for the entire system, summed over all links.

Note that the HMC bringup (which happens when the sample software calls ReadRam) can take awhile. This initialization is only required once after programming the FPGA.

Simulation

The following explains how to simulate the GUPS application using the HMC controller. Rather than provide a simulator, test benches work with all major simulation packages. In particular, Altera's Questa simulator is recommended for simulating designs that can communicate with the HMC controller.

Also, although it is required to simulate the HMC, the HMC Bus Functional Model (BFM) is not provided with this package. To obtain the BFM, contact Micron's HMC support via micron.com and request the r22107 BFM.

Apply the patch in hmc_release/hmc_bfm:

1. *cd hmc_release/hmc_bfm*
2. *patch pico-change-r22107.patch*

A few different simulation tests are provided for users to explore. The first is a very simple and clean way to see the default GUPS test, whereas the second is a more complex version that runs multiple GUPS tests (with different runtime parameters) in the same run.

All simulation test benches are distributed in *hmc_release/test*, with one test per sub-directory in that folder. See *hmc_release/test/README.markdown* for instructions on how to run the simulation.

Before running a simulation, users should set the following environment variables:

- PICO_REPO: This should point to the root of your Pico Framework installation, which is likely at /usr/src/picocomputing_<version>/
- HMC_REPO: This should point to the root of your HMC controller release.

To run a test, first set the aforementioned environment variables. Then cd into the desired test directory and run:

../compile_test.sh && ../run_test.sh

This compiles the HMC controller, BFM, and GUPS application code for simulation. It will also run the simulation, as defined by *sim_top.v* within your test sub-directory. Some error checking is done when the simulation completes, and then a pass or fail message is printed. When successful, you should see the following printed near the end of the simulation:

 # [992162000] Test Result: SUCCESS

*# ** Note: $stop : ../sim_top.v(87)*

Performance

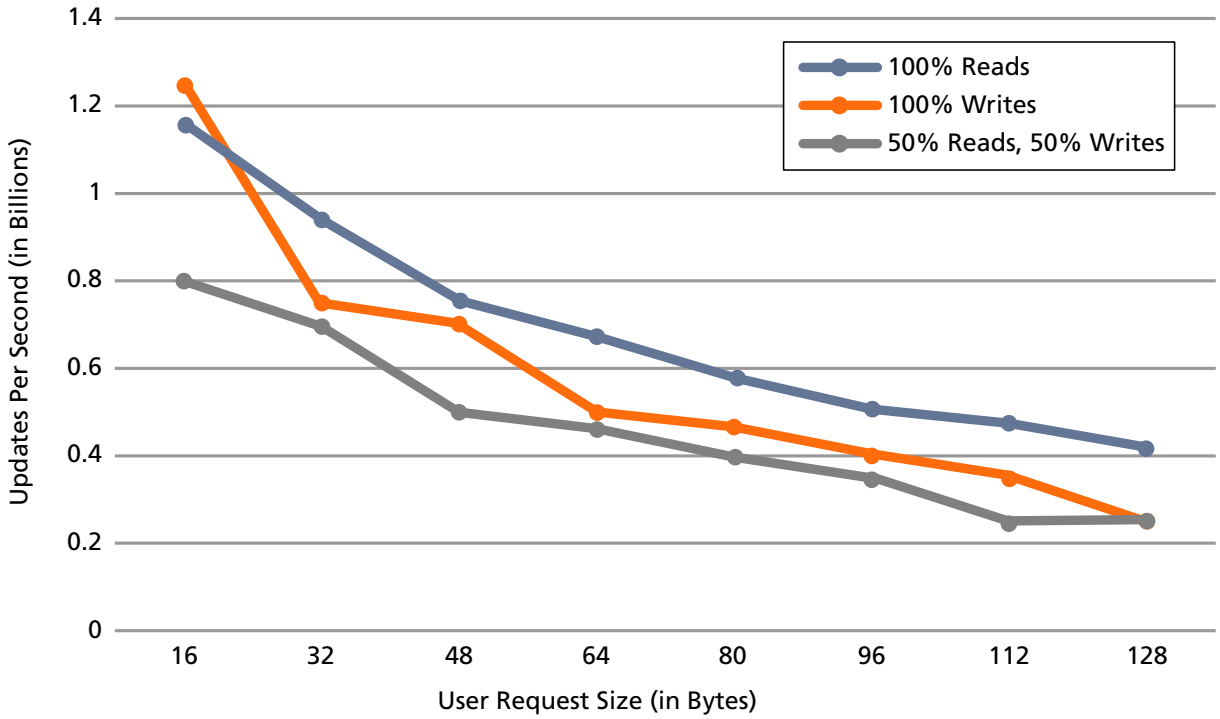
HMC controller performance is measured using a distributed example application on supported evaluation systems. Results are categorized as follows:

- Random Accesses: Measures the number of possible READ and WRITE operations per second. For the read-modify-write mode, an *access* is determined as one read followed by one write. Therefore, the total number of operations in read-modify-write mode is 2x the performance reported.
- Bandwidth: Measures how much data can be read from and/or written to the HMC. Packet overhead is not included in this measurement. For read-modify-write mode, read bandwidth is equal to write bandwidth because for each READ operation there is a corresponding WRITE operation. Therefore read bandwidth is reported only when operating in read-modify-write mode.
- Link Efficiency: Measures how efficiently the controller utilizes the maximum theoretical bandwidth of the links. To compute TX efficiency, the write bandwidth is divided by the maximum theoretical bandwidth on the TX link. Read bandwidth is used to compute RX link efficiency in a similar manner.

Results are stated in terms of bandwidth and request rate to emphasize the relationship between the two. For example, while the system operates within the maximum bandwidth range and up to the point when the maximum bandwidth is approached, the request rate remains the same regardless of packet size. At the point when the maximum bandwidth is approached, the request rate can increase if the request sizes are reduced.

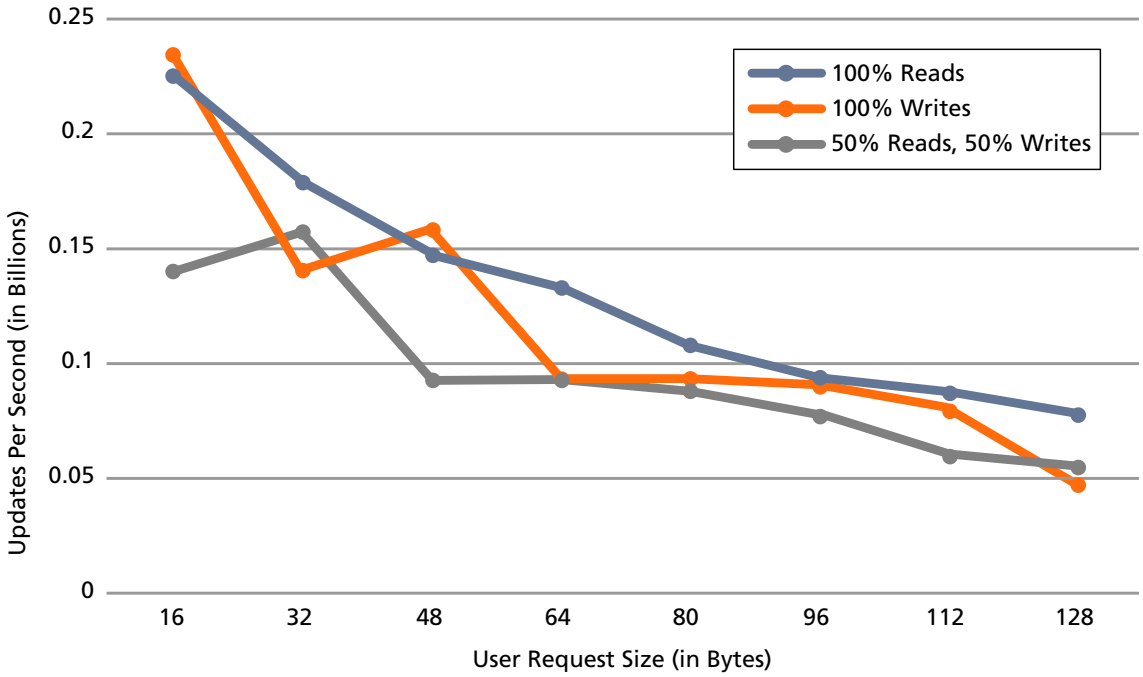
As expected, this graph shows that the number of memory accesses is inversely related to the request size. Shown starting at the left of the graph, HMC with the Micron HMC controller is capable of performing over a billion READ and WRITE operations per second when the request size is small. A small reduction is evident during the read-modify-write mode, as opposed to during read-only mode.

Figure 14: Random Access Performance – SB-800



When comparing performance of the AC-510 with that of the SB-800, a reduction to 1/8th the performance might be expected: The one half-width link of AC-510 as opposed to the four full-width links of SB-800 would support this expectation. However, the random access performance is between 1/5th and 1/6th that of the SB-800. Therefore, if users care only about random access performance, using half-width links, instead of full-width links may be more efficient.

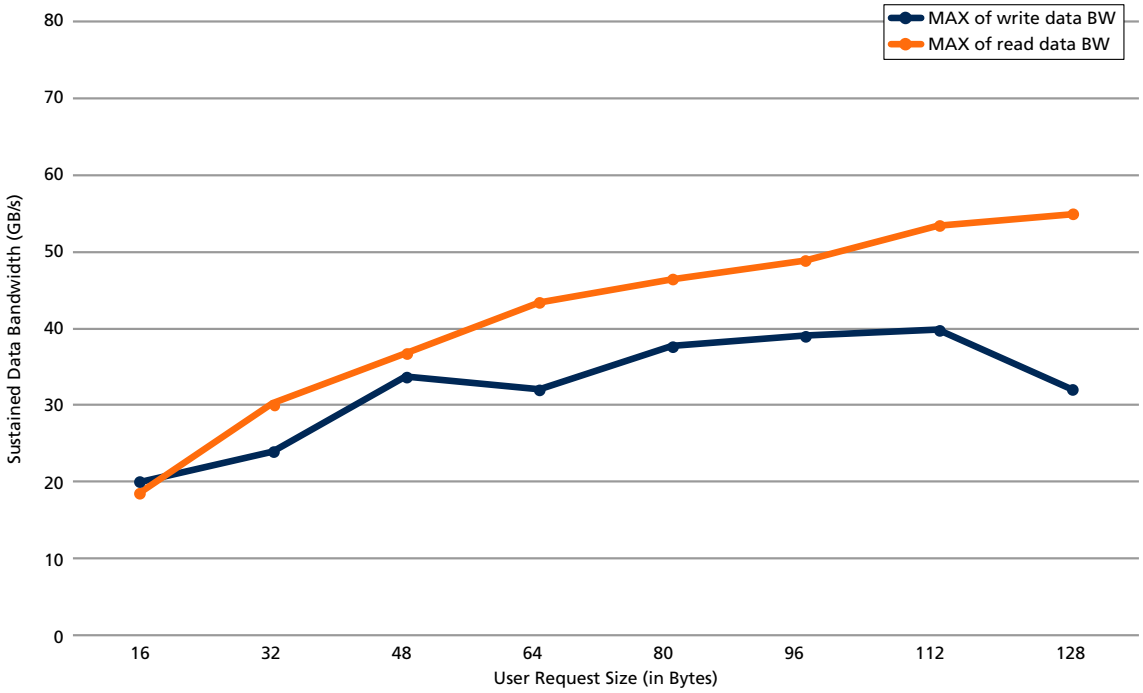
Figure 15: Random Access Performance – AC-510



Bandwidth

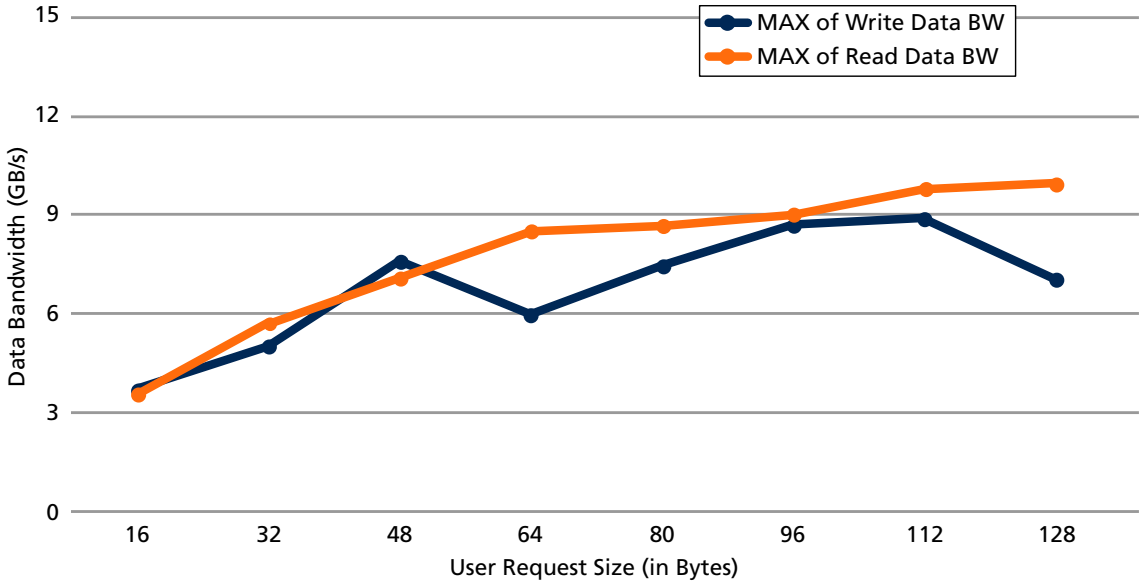
Bandwidth is computed from performance measurements. Particularly important is sustained read and write data bandwidth when operating in the respective read-only and write-only modes as shown below. Measured on SB-800, this data bandwidth in and out of an HMC is impressive when running on more than one link.

Figure 16: Bandwidth – SB-800



On the AC-510, maximum sustained bandwidth is approximately equal to that of high-speed DDR3 memories. One advantage still maintained over traditional DDR memories is the full-duplex bus.

Figure 17: Bandwidth – AC-510



Link Efficiency

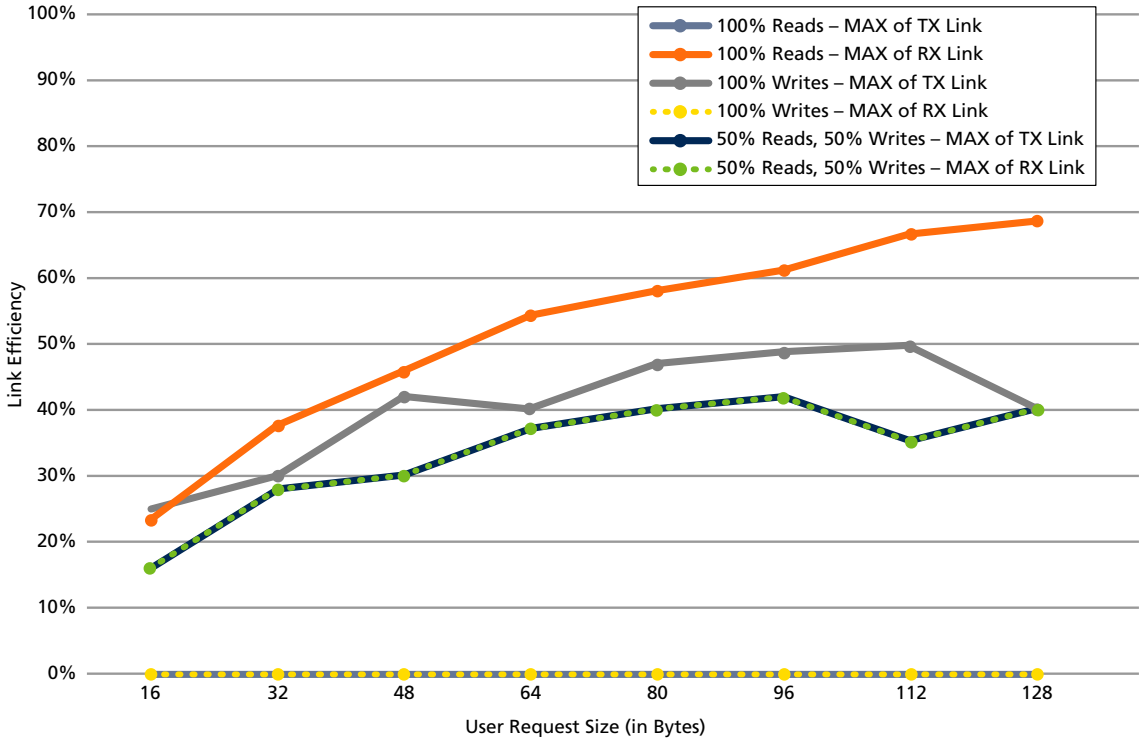
The HMC protocol requires one FLIT of overhead for each write request (on the TX link) and two FLITS of overhead for each read request (one on the TX link, one on the RX link). The maximum request size is 128B (8 FLITS). Therefore, maximum theoretical link efficiencies broken down by operating mode are as follows:

- Write-only: 89% on TX link, 0% on the RX link
- Read-only: 0% on the TX link, 89% on the RX link
- Read-modify-write: 80% on the TX link, 89% on the RX link

With these peak theoretical efficiencies, as limited by the protocol, the efficiency of the TX and RX links as measured on the SB-800 are shown in the figure below. Small request sizes do not use the available link bandwidth efficiently, primarily because the request size is small yet the overhead is large. Large request sizes use bandwidth more efficiently because more bandwidth is used to move data instead of being used for overhead.

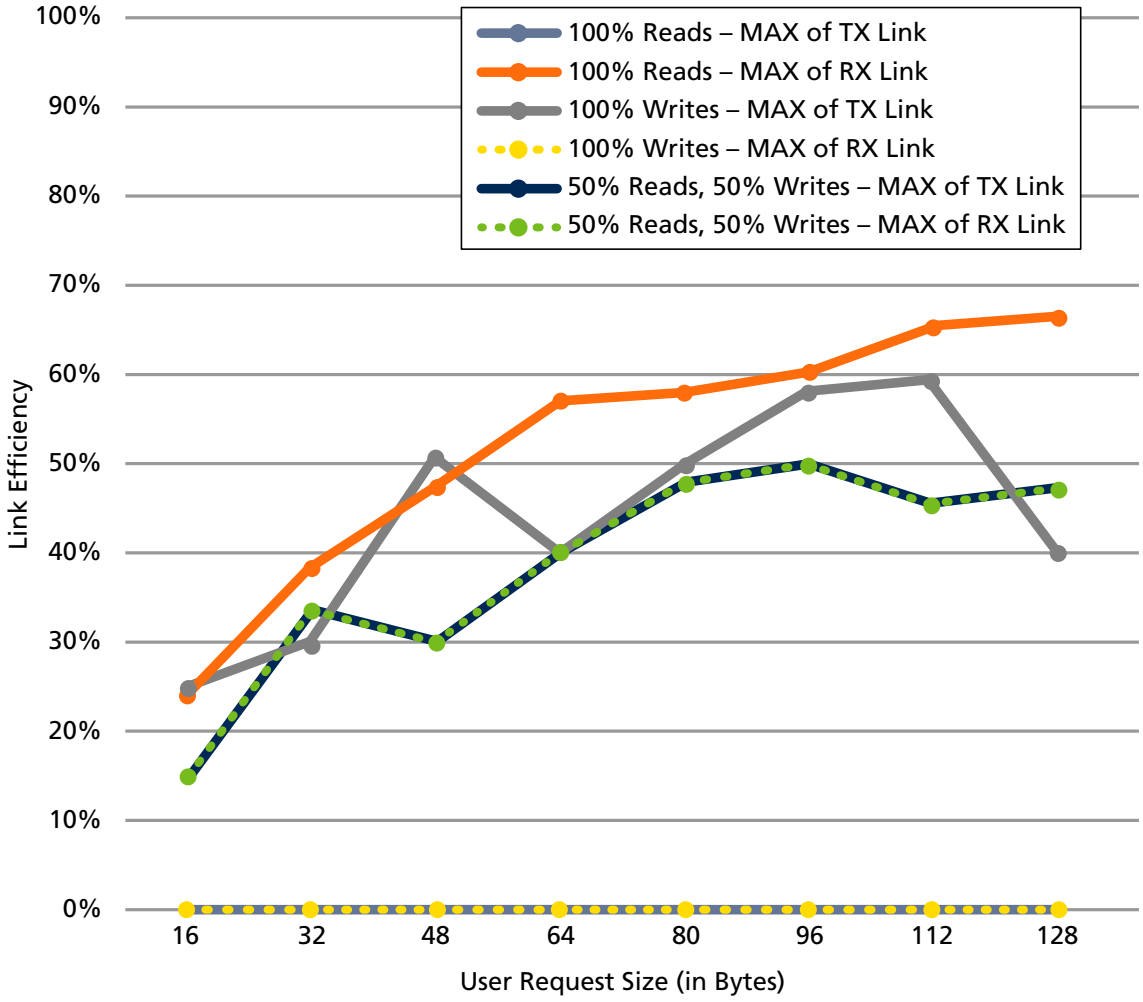
> **Note:** The write-only operating mode at 100% does not use the RX link; the read-only operating mode at 100% uses the TX link minimally.

Figure 18: SB-800 Efficiency – TX/RX Links



The graph below shows the same link efficiency calculation for the AC-510. Like the SB-800, read-only mode is very efficiently using the available bandwidth on the RX link for large request sizes.

Figure 19: AC-510 Efficiency – TX/RX Links



Resource Utilization

The size of the HMC controller IP varies depending on its configuration and the FPGA device type. The Altera/Xilinx toolchain produces information to create a configuration report, as shown in the table below. The table refers to a controller with five user ports.

Table 10: Configuration Reports

Evaluation System	ALM/LUT (K)	Registers (K)	Memory Block (Mb)
SB-800	47	79	1.79
AC-510	27	49	4.06

Appendix A

ASIC-Required Low-Level Signals

This Appendix describes the requirements for incorporating the IP from this document into a user's ASIC.

The required low-level signals for the HMC controller IP to communicate with the HMC are well-defined in the Micron HMC Gen2 data sheet, refer to that document for more information about these signals.

Table 11: Low-Level Signals

Signal Name	Type	Width (Bits)	Description
lxrxp[n:0]	Input	–	Receiving lanes
lxrxn[n:0]			
lxrxps	Input	–	Power-reduction input
lxtxp[n:0]	Output	–	Transmitting lanes
lxtxn[n:0]			
lxtxps	Output	–	Power-reduction output
ferr_n	Input	1	Fatal error indicator. HMC drives LOW if fatal error occurs, otherwise the pulldown is turned off and floats HIGH. ferr_n operates in V_{DDK} domain. Requires external pull-up resistor. ferr_n can be wire-OR'd among multiple HMC devices. $R_{ON} = 300\Omega$; $R_{OFF} = 10k\Omega$
sda	I/O	1	I^2C SDA to the HMC.
scl	I/O	1	I^2C SCL the HMC.

Note: 1. Directionality in the interface signal tables shown here is with respect to the HMC controller.

The Micron HMC controller builds on the Pico Computing Framework, which includes a memory-mapped bus called the PicoBus. An ASIC version of this controller must include the ports for the PicoBus as shown in the following table.

Table 12: Standard Pico Framework Signals

Signal Name	Type	Width (Bits)	Description
extra_clk	Input	1	Slow clock (100 MHz).
PicoClk	Input	1	PicoBus clock (4 MHz).
PicoRst	Input	1	PicoBus synchronous reset.
PicoAddr	Input	32	PicoBus address.
PicoDataIn	Input	32	PicoBus write data.
PicoRd	Input	1	PicoBus read strobe.
PicoWr	Input	1	PicoBus write strobe.
PicoDataOut	Output	32	PicoBus read data.

Contact Micron for more information regarding the Pico Framework signals, or for incorporating this IP into an ASIC.

Revision History

Rev. A – 1/16

- Initial release

8000 S. Federal Way, P.O. Box 6, Boise, ID 83707-0006, Tel: 208-368-4000
www.micron.com/products/support Sales inquiries: 800-932-4992