Introduction

This application note introduces the dedicated PCI Express logic block implemented in Arria® II GX FPGA hardware and describes the following:

- The hard IP implementation of the PCI Express MegaCore® in the Arria II GX device
- The DDR2 SDRAM High-Performance Controller
- The PCI Express protocol
- Implementing the PCI Express MegaCore with the Quartus® II software
- Verification of multiple IP modules with the ModelSim® software

PCI Express is a point-to-point high-speed serial I/O interface that offers the ability for components to communicate with a much higher level of efficiency. This design conforms to the PCI Express Base Specification, Rev 2.0.

This design represents a sample interface between:

- A device that drives commands to the Arria II GX FPGA using the PCI Express protocol (Root Complex)
- An Arria II GX FPGA (Endpoint)
- An external DDR2 SDRAM memory.

The Altera® PCI Express-to-DDR2 reference design is an example of a typical user application that interfaces to an Altera PCI Express MegaCore function.

General Description

In this design, a Root Complex interfaces with the FPGA Endpoint. The FPGA Endpoint interfaces with external DDR2 SDRAM (refer to Figure 1).

Figure 1. Root Complex, FPGA Endpoint, and External DDR2 SDRAM
The reference design implements the Endpoint on the FPGA and includes the following blocks:

- A PCI Express MegaCore that interfaces with the Root Complex
- A DDR2 SDRAM High-Performance Memory Controller to interface to the DDR2 SDRAM memory
- Logic to interface between the PCI Express MegaCore and the DDR2 SDRAM High-Performance Memory Controller

The Root Complex issues read and write commands to the Endpoint across the PCI Express link. The Endpoint can also initiate reads and writes to the Root Complex through direct memory access (DMA) reads and writes. This is accomplished through read and write transactions.

**Transaction Types**

Transactions occur when packets are sent from one device to another. A packet consists of a header and data. The header contains information about the type of transaction, the data contained, and other pertinent details regarding the packet. The data is the information to be transferred between the components. In this case, the components are the Root Complex, the Endpoint, and the DDR2 SDRAM. The packets in this case are transaction layer packets (TLPs).

This reference design handles the following TLPs:

- Memory Read Request (MRd)
- Memory Write Request (MWr)
- Completion with Data (CplD)

The initiating device can send a MRd to the target device that answers with a CplD, which contains the requested data. The initiating device can also send a MWr that contains data.

The following four types of transactions are initiated across the PCI Express link between the Root Port and the FPGA Endpoint.

- Root-Complex-Initiated Memory Write
- Root-Complex-Initiated Memory Read
- Endpoint-Initiated DMA Write
- Endpoint-Initiated DMA Read
Root-Complex-Initiated Memory Write
The Root Complex sends data to the Endpoint by initiating a write transaction. This transaction is a MWr-type transaction and is a posted transaction, which means that there is no return communication (refer to Figure 2).

Figure 2. Root-Complex-Initiated Memory Write

The following events occur in Figure 2:
1. The Root Complex sends the MWr transaction to the FPGA.
2. The PCI Express hard IP receives the packet and performs all physical layer and data link layer services and also performs the flow control services in the transaction layer.
3. The PCI Express hard IP provides the application logic with a TLP in the format defined in the PCI Express Compiler User Guide.
4. The application logic decodes the TLP and sends the data to the DDR2 SDRAM High-Performance Memory Controller.
5. The data is written to the DDR2 SDRAM memory.

Root-Complex-Initiated Memory Read
The Root Complex requests data from the Endpoint by initiating a read request transaction. This is a MRd-type transaction and is a non-posted transaction, which means that there is return communication in the form of a completion with data (CplD)-type transaction (refer to Figure 3).
The following events occur in Figure 3:

1. The Root Complex sends the MRd transaction to the FPGA.
2. The PCI Express hard IP receives the packet and performs all physical layer and data link layer services. It also performs the flow control services in the transaction layer and provides the application logic with a TLP in the format defined in the PCI Express Compiler User Guide.
3. The application logic decodes the TLP and initiates a memory read request from the DDR2 SDRAM High-Performance Memory Controller.
4. The DDR2 SDRAM memory obtains the data through the DDR2 SDRAM High-Performance Memory Controller.
5. The application logic creates the header for the data and a completion with data TLP is assembled and sent to the PCI Express hard IP for transmission.
6. The PCI Express hard IP performs all the remaining transaction layer services and performs all the data link layer and physical layer services.
7. The FPGA sends the CplD packet to the Root Complex.

**Endpoint-Initiated DMA Write**

The Endpoint sends data to the Root Complex by initiating a write transaction, referred to as a DMA write. This is a MWr-type transaction and is a posted transaction, which means that there is no return communication (refer to Figure 4).
Figure 4. Endpoint-Initiated DMA Write

The following events occur in Figure 4:

1. The application logic reads the DMA registers to determine whether a write request transaction to the Root Complex is waiting to occur. The application logic then sends a read request to the DDR2 SDRAM High-Performance Controller for the data to send.

2. The data is obtained from the DDR2 SDRAM memory through the DDR2 SDRAM High-Performance Memory Controller.

3. The header for the data is created and a memory write TLP is assembled in the application logic and sent to the PCI Express hard IP for transmission.

4. The PCI Express hard IP performs all remaining transaction layer services and performs all data link layer and physical layer services.

5. The FPGA sends the MWr packet to the Root Complex.

Endpoint-Initiated DMA Read

The Endpoint requests data from the Root Complex by initiating a read transaction, referred to as a DMA read. This is a MRd-type transaction and is a non-posted transaction, which means that there is return communication in the form of a CplD-type transaction (Figure 5).
The following events occur in Figure 5:

1. The application logic reads the DMA registers to determine whether a read request transaction to the Root Complex is waiting to occur. The application logic then assembles a memory read TLP and sends it to the PCI Express hard IP for transmission.

2. The PCI Express hard IP performs all remaining transaction layer services and performs all data link layer and physical layer services.

3. The FPGA sends a MRd packet to the Root Complex.

4. The Root Complex sends a CplD packet to the FPGA.

5. The PCI Express hard IP receives the completion packet and performs all physical layer and data link layer services. It also performs the flow control services in the transaction layer.

6. The PCI Express hard IP provides the application logic a completion with data TLP in the format defined in the *PCI Express Compiler User Guide*.

7. The application logic decodes the TLP and sends the data to the DDR2 SDRAM High-Performance Memory Controller.

8. The data is written to the DDR2 SDRAM memory.

**Reference Design Overview**

Figure 6 shows a high-level block diagram of the reference design. The reference design comprises a PCI Express MegaCore function, a DDR2 SDRAM controller, and the reference design application logic between the two. This sample application logic includes a TX application logic block (labeled “tx_top”), and an RX application logic block (labeled “rx_top”).
This section provides an overview of the following reference design components:

- **PCI Express MegaCore Function**—Transmits and receives data to and from the Root Complex through a high-speed serial link.
- **DDR2 SDRAM Controller**—Transmits and receives data to and from the DDR2 SDRAM external memory.
- **RX Application Logic**—Interfaces with the PCI Express MegaCore and the DDR2 SDRAM controller to receive data.
- **TX Application Logic**—Interfaces with the PCI Express MegaCore and the DDR2 SDRAM controller to transmit data.
- **DMA Engine**—Monitors the DMA registers to initiate transactions. Logic spread through the TX application logic and the RX application logic.

The logic has a different data flow for the TLP header and the TLP data. TLP headers are either decoded in the RX application logic or assembled in the TX application logic. Information from the headers are extrapolated into commands. The commands are generally used in the logic and stored in 128-bit FIFOs. This is to accommodate a four-double-word (DW) TLP header (128-bits) or a three-DW header (96-bits). A one DW is 32-bits.
PCI Express MegaCore Function

The PCI Express MegaCore function connects to the system application logic on one side. On the other side, it connects to the PHY that connects to the PCIe link. The PCI Express MegaCore function performs the PCIe functions of the physical layer, data link layer, and transaction layer, as specified in the *PCI Express Base Specification Rev. 2.0*.

This reference design instantiates the PCI Express MegaCore in the hard IP implementation with the Avalon®-ST interface. The core is instantiated using the MegaWizard™ Plug-In Manager flow.

The PCI Express MegaCore in the hard IP implementation handles the specifications for the physical layer, data link layer and the flow control for the transaction layer (refer to Figure 7).

**Figure 7.** PCI Express MegaCore in the Hard IP Implementation

The output of the transaction layer are packets that follow the PCI Express protocol and are referred to as TLPs. A TLP is divided into header and data info. TLPs are broken into double words (DW) that are either part of the header or data (refer to Figure 8).

**Figure 8.** TLP Header and Data Information

<table>
<thead>
<tr>
<th>Header</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>H0 ... Hx</td>
<td>D0, D1, ... Dy</td>
</tr>
</tbody>
</table>
TLPs can have a three-DW or four-DW header. The payload length is specified in the TLP header in the length field in the DWs (refer to Figure 9).

**Figure 9. Double-Word Header and Payload Length**

<table>
<thead>
<tr>
<th></th>
<th>+0</th>
<th>+1</th>
<th>+2</th>
<th>+3</th>
</tr>
</thead>
<tbody>
<tr>
<td>H0</td>
<td>0</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>H1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>H2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>H3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The TLPs are delivered to the application layer that is implemented in the FPGA logic. The TLPs are in the following form and are accompanied by the control signals shown in Figure 10. The TLP information is on data. Each beginning of a TLP is accompanied by a start-of-packet (SOP). An end-of-packet (EOP) signifies the end of a TLP and the Valid signal is high for the duration that data must be clocked and contains TLP information (refer to Figure 10).

**Figure 10. TLPs Delivered to the Application Layer**

The nature of the TLPs is described in section 2.1 of the *PCI Express Base Specification, Rev 1.1* and in section 5 of the *PCI Express Compiler User Guide*.

The PCI Express core reserves system address space that can have up to eight base address registers (BARs) that reference it. Each BAR stores the starting address of the memory segment being referenced. A standard PCI Express application generally uses BAR0 and BAR1 as the standard reference for the link. You can specify additional BARs for alternative uses.

In this design, BAR0 and BAR1 are used to reference the DDR2 SDRAM memory. BAR2 is used to reference the DMA registers that are monitored by the application logic. This allows the Root Complex to write to the DMA registers by writing to BAR2. Therefore, it is possible to initiate Endpoint transactions from the Root Complex. That is how this design writes to the DMA registers.
**DDR2 SDRAM Controller**

The DDR2 SDRAM controller interfaces with the TX and RX application logic on the local side and the off-chip DDR2 SDRAM memory on the system side. The controller uses interface signals to communicate with the RX and TX application logic.

When writing to the DDR2 SDRAM memory, the RX application logic (rx_ddr) asserts the `local_write_req` signal while supplying addresses to be written on the `local_address` signal. The DDR2 SDRAM High-Performance Memory Controller then asserts `local_wdata_req` and within one clock cycle the rx_ddr module supplies the data on the `local_wdata` signal (refer to Figure 11).

**Figure 11.** Writing to the DDR2 SDRAM Memory

When reading from the DDR2 SDRAM memory, the RX application logic (rx_ddr) asserts the `local_read_req` signal while supplying addresses to be read from on the `local_address` signal. After some time, the DDR2 SDRAM High-Performance Memory Controller returns the data to the TX application logic (tx_ddr_resp) on the `local_rdata` signal and asserts the `local_rdata_valid` signal (refer to Figure 12).

**Figure 12.** Reading from the DDR2 SDRAM Memory
RX Application Logic

The RX application logic block receives TLP packets from the Root Complex through the PCI Express MegaCore. The RX application logic block decodes the TLP packets and, based on the type of transaction, either writes data to the DDR2 memory or sends read request information to the TX application logic block so that it can grab information from the DDR2 memory to complete the request. The RX application logic block also holds the DMA registers that are monitored by the TX application logic.

RX Application Logic Components

The RX application logic comprises the following components:

- RX PCIe Packet Receive Block (rx_pcie)
- RX Command FIFO Buffer (72 × 32)
- RX Data FIFO Buffer (72 × 128)
- Pending Read FIFO Buffer (50 × 32)
- RX DDR2 Write Block (rx_ddr)

The rx_pcie module interfaces with the PCI Express MegaCore and the rx_ddr module interfaces with the DDR2 SDRAM High-Performance Controller. The FIFO buffers are monitored by the components and hold the data transferred between the components. The FIFO buffers also synchronize the data crossing the PCIe and the DDR2 clock domains.

RX PCIe Packet Receive Block (rx_pcie)

The rx_pcie block receives TLPs from the PCI Express MegaCore function. It decodes the system-side signals from the PCI Express MegaCore function. Based on the type of request, the rx_pcie block decides whether to write data to the DDR2 SDRAM, submit a read request to the pending read FIFO, or write to the DMA registers.

Packets are accompanied by the start of packet (SOP), end of packet (EOP), and the data valid (Valid) signals. The header is extracted and if there is data, it is written to the RX Data FIFO buffer.
Data can be aligned or non-aligned. Aligned data means that the data, starting with \( D_0 \), starts on the lower 32 bits of the 64-bit bus (refer to Figure 13).

**Figure 13.** RX PCIe packet Receive Block (rx_pcie)—Aligned

Non-aligned data means that \( D_0 \) starts on the higher 32 bits of the 64-bit bus (refer to Figure 14).

**Figure 14.** RX PCIe packet Receive Block (rx_pcie)—Non-Aligned
To determine if the data is aligned or non-aligned, examine bit 2 of the address. If address[2] is 0, the data is aligned. If address[2] is 1, the data is non-aligned (refer to Figure 15).

**Figure 15. Aligned and Non-Aligned Data**

The rx_pcie module manages access to and monitors the DMA registers. A received transaction targets the DMA registers if it targets BAR2. When directed by the dma_start signal, and when it is ready to start the DMA transaction, the rx_pcie block uses the information from the DMA registers to generate one or more requests that are then loaded into the RX command FIFO buffer and the RX pending read FIFO buffer.

**RX Command FIFO Buffer (72 × 32)**

The RX command FIFO buffer receives the reformatted commands from the rx_pcie block. The rx_ddr block monitors the RX command FIFO buffer and, if it is not empty, reads the commands. The RX command FIFO buffer also converts the commands from the PCI Express clock domain to the DDR2 clock domain and transfers the commands to the rx_ddr block.

**RX Data FIFO Buffer (72 × 128)**

The RX Data FIFO buffer receives data from the rx_pcie block to be transferred to the DDR2 SDRAM memory. The RX DDR2 block, rx_ddr, reads the data from the RX Data FIFO buffer.

The upper eight bits from the FIFO buffer are byte-enables for the 64-bit data bus. The read operations from the RX Data FIFO buffer always occur in trunks of 72 bits with only 64 bits stored in the DDR2 SDRAM memory.
Pending Read FIFO Buffer (50 x 32)
When receiving a read request from the PCI Express MegaCore function, the RX application logic also receives details about how much completion data to expect from the DDR2 SDRAM memory and where to send the read completion data when ready. The RX application logic stores the request details received by the RX command FIFO buffer in the pending read FIFO buffer. The tx_ddr_resp block reads from the pending read FIFO buffer and uses the information to track the read completion data coming back from the DDR2 SDRAM controller.

RX DDR2 Interface Logic (rx_ddr)
The RX DDR2 interface logic draws commands from the RX command FIFO. Based on the RX command FIFO information, the rx_ddr block generates the DDR2 controller signals to request DDR2 read and write requests. For DDR2 write requests, the rx_ddr block works with the TX read tag RAM to determine where to write the data in the DDR2 SDRAM memory.

TX Application Logic
The TX application logic block is responsible for generating TLP packets to transmit to the Root Complex using PCI Express. It also controls the DDR2 SDRAM controller interface and receives data from the DDR2 SDRAM. In addition, it interacts with the DMA registers to generate TX DMA reads and writes to send to the Root Complex using PCI Express.

TX Application Logic Components
The TX application logic comprises the following components in tx_top:
- TX DDR2 Read Block (tx_ddr_resp)
- TX DMA Request Generation Block (tx_dma_rd)
- TX Command FIFO Buffer (128 x 32)
- TX Data FIFO Buffer (64 x 128)
- TX Read Tag RAM
- TX PCIe Packet Generation Block (tx_pcie)
- Read Bypass FIFO Buffer (128 x 32)

The tx_pcie module interfaces with the PCI Express MegaCore; the tx_ddr_resp module interfaces with the DDR2 SDRAM High-Performance Controller, and the tx_dma_rd module monitors the DMA registers. The FIFO buffers are monitored by the components and hold the data transferred between the components. The FIFO buffers also synchronize the data crossing the PCIe and DDR2 clock domains.

TX DDR2 Read Block (tx_ddr_resp)
The TX DDR2 read response state machine (tx_ddr_resp) checks the RX pending read FIFO buffer for pending read requests. If there is a pending read request, tx_ddr_resp uses the information from the RX pending read FIFO to determine how much data to expect from the DDR2 SDRAM controller. It also determines when and where to send the DDR2 data when available. The tx_ddr_resp module also monitors the DDR2 SDRAM controller system-side data bus and the data valid signal.
TX DMA Request Generation Block (tx_dma_rd)

The TX DMA request generation block (tx_dma_rd) monitors the DMA registers that are in the RX application logic. When directed by the dma_start signal, and when ready to start the DMA transaction, the tx_dma_rd block uses the information from the DMA registers to generate one or more TX requests and puts them in the TX command FIFO.

TX Command FIFO Buffer (128 × 32)

The TX command FIFO buffer stores commands generated by the tx_dma_rd block and the tx_ddr_resp blocks. Commands are in the PCIe TLP header format and have a width of 128 bits to hold up to a four-DW header. If the header is a three-DW header, the last DW is considered reserved.

The tx_pcie block receives the PCIe commands from the TX command FIFO buffer after they cross from the DDR2 clock domain to the PCIe clock domain.

TX Data FIFO Buffer (64 × 128)

The TX data FIFO buffer provides buffer space for the data transferred from the DDR2 SDRAM controller to the PCI Express MegaCore function. During the root-complex-initiated read completions, or during Endpoint-initiated write requests, the DDR2 SDRAM controller returns requested data, which enters into this FIFO buffer.

The TX DDR2 response control block, tx_ddr_resp, controls the write operations to this buffer. The TX PCIe control block, tx_pcie, controls the read operations from this buffer.

The TX data FIFO buffer size is 64 × 128 bits to ensure that it can store all the DWs in the TLP, if required, before sending the packet to the tx_pcie PCIe controller. Unlike the RX Data FIFO buffer, this FIFO buffer always operates as a 64 bit wide buffer. The read or write from this FIFO buffer always occurs 64 bits at a time.

TX Read Tag RAM

The TX read tag RAM stores the TX read requests information. Based on the stored information, it works with the rx_ddr block to determine where to send the read completion data when it is ready to be written to the DDR2 SDRAM.

TX PCIe Packet Generation Block (tx_pcie)

The TX PCIe packet generation block monitors the TX command FIFO and the TX read bypass FIFO. If the FIFOs are not empty, the tx_pcie block draws commands from them. Based on the command information, the tx_pcie block generates the appropriate TLP to initiate transactions and sends the TLP to the PCI Express MegaCore.

The commands in the command FIFO and the read bypass FIFO are in the TLP header format, as specified by the PCI Express Base Specification, Rev 2.0.
In addition to generating the TLP, the \texttt{tx_pcie} block also generates the start of packet (\texttt{tx_st_sop}), the end of packet (\texttt{tx_st_eop}), and the Data Valid (\texttt{tx_st_valid}) signals that accompany the TLP (refer to Figure 16).

**Figure 16.** TX PCIe Packet Generation Block (\texttt{tx_pcie})

In this application, the TLPs generated are memory reads, memory writes, and completion packets. Data can be aligned or non-aligned. Aligned data means that the data, starting with \texttt{D0}, starts on the lower 32 bits of the 64-bit bus (refer to Figure 17).

**Figure 17.** TX PCIe Packet Generation Block—Aligned Data
Non-aligned data means that $D_0$ starts on the higher 32 bits of the 64-bit bus (refer to Figure 18).

**Figure 18. TX PCIe Packet Generation Block—Non-Aligned Data**

To determine if the data is aligned or non-aligned, examine bit 2 of the address. If $address[2]$ is 0, the data is aligned. If $address[2]$ is 1, the data is unaligned (refer to Figure 19).

**Figure 19. Aligned and Non-Aligned Data**
**Read Bypass FIFO Buffer (128 × 32)**

The read bypass FIFO buffer provides buffer space for TX read requests when a lack of transmit credits prevent the tx_pcie block from sending TX read requests to the PCI Express MegaCore function. When the PCI Express MegaCore function is not ready to accept TX read requests, the read bypass FIFO buffer can temporarily store the read requests, which enables the tx_pcie block to send other requests in the meantime.

This is an optimization, not a requirement because the PCI Express transaction layer handles the flow control of packets. This is only to optimize the system by allowing the application logic to send other types of TLPs if read requests are not being accepted.

**DMA Engine**

The purpose of the DMA engine is to allow the Endpoint to be an active component that drives transactions to the Root Complex. In this design, the Root Complex writes to the DMA registers by sending transactions over the PCI Express link targeting BAR2. The registers are located in the RX application logic. When DMA drives the transactions, it coordinates transactions to and from the DDR2 SDRAM memory.

Setting up the DMA registers in the DMA engine initiates DMA transactions. The DMA registers are located in the rx_pcie module. These registers are memory-mapped to base address register BAR2 in the PCI Express MegaCore function. Target transactions can access the registers through their memory-mapped addresses. DMA registers require the Root Complex to supply their contents and consist of the following elements:

- PCIe lower address register (0×0)
- PCIe upper address register (0×4)
- Byte counter register (0×8)
- DMA control and status register (0×C)
- DMA local address register (0×10)

**Functional Description**

This section describes operations of the various modules in the PCI Express-to-DDR2 SDRAM reference design and provides information about the following topics:

- Transactions Initiated from the Root Complex
- Transactions Initiated from the Endpoint
- Memory Mapping
- DMA Register Access
Transactions Initiated by the Root Complex

After the PCI Express MegaCore function receives a packet successfully and without errors, it sends the packet to the PCIe RX control logic, rx_pcie. The rx_pcie logic decodes the PCIe packet, receives any associated data, then forwards the request information into the RX command FIFO buffer and the data into the RX Data FIFO buffer.

The RX control logic on the DDR2 side, rx_ddr in Figure 6 on page 7, receives the request and the data from the RX FIFO buffers, splits the request into multiple DDR2 requests, and sends the requests to the DDR2 SDRAM controller.

This section describes the following topics:

- Root-Complex-Initiated Memory Read
- Root-Complex-Initiated Memory Write

Root-Complex-Initiated Memory Read

When a memory read request reaches the RX application logic, the following sequence of actions occurs:

1. The PCI Express MegaCore function asserts rx_st_sop, and H0 and H1 of the header are available on rx_st_data. The RX application logic looks at the format (Fmt) and type (Type) fields. The transaction type can be determined to be a memory read.
2. The rx_sop signal is de-asserted, and H2 and H3, containing the address information, are available on rx_st_data.
3. The rx_pcie module extracts the necessary information, reformats the request, and puts the request into the command FIFO.
4. In the DDR2 clock domain, the rx_ddr logic accepts the read request from the command FIFO and translates it into one or more DDR2 read requests. The rx_ddr logic sends the read requests to the DDR2 SDRAM controller sequentially.
5. The DDR2 SDRAM controller reformats the read requests and sends them to the DDR2 SDRAM memory.
6. After some time, the DDR2 SDRAM memory returns the read request data.
7. The TX data FIFO accumulates the data. The tx_ddr_resp logic monitors this action and determines when to return a read completion packet to the PCI Express MegaCore function.
8. When enough data accumulates in the TX data FIFO, tx_ddr_resp generates a PCIe read completion packet and puts it in the TX command FIFO.
9. Through the TX data FIFO, the packet crosses the clock domain and the tx_pcie logic reads it as a PCIe TX packet.
10. If the PCI Express MegaCore function is ready to accept the read completion packet, tx_pcie transmits the generated TLP and asserts the tx_st_sop and tx_st_valid signals.
11. As the last DW of header or data is sent, tx_pcie asserts the tx_st_eop signal.
Root-Complex-Initiated Memory Write

When a target memory write request reaches the RX application logic, the following sequence of actions occurs:

1. The PCI Express MegaCore function asserts rx_st_sop, and H0 and H1 of the header are available on rx_st_data. The RX application logic looks at the format (Fmt) and type (Type) fields. The transaction type can be determined to be a memory write.

2. The rx_sop signal is de-asserted, and H2 and H3, containing the address information, are available on rx_st_data.

3. The application logic extracts the necessary information, reformats the request, and puts the request into the command FIFO.

4. In the DDR2 clock domain, the rx_ddr logic accepts the write request from the command FIFO and translates it into one or more DDR2 write requests. The rx_ddr logic sends the write requests to the DDR2 SDRAM controller sequentially.

5. When the DDR2 SDRAM controller is ready to accept the write data, it asserts the local_wdata_req signal. The RX application logic responds by providing the write data on the local_wdata bus.

6. The DDR2 SDRAM controller reformats the write requests and sends them to the DDR2 SDRAM memory.

Transactions Initiated from the Endpoint

After the PCI Express MegaCore function receives a packet targeting BAR2 successfully, signifying instructions for a DMA read or write, it sends the packet to the PCIe RX control logic, rx_pcie. The rx_pcie logic decodes the PCIe packet and writes the request to the DMA registers.

The RX control logic asserts the dma_start signal that is read by the tx_dma_rd block.

This section describes the following topics:

- Endpoint-Initiated DMA Read
- Endpoint-Initiated DMA Write

Endpoint-Initiated DMA Read

To initiate a DMA read request, the Root Port first programs all the DMA registers sequentially. After the DMA control register is written to and decoded, the dma_start signal asserts for one clock cycle to initiate the DMA read.

When processing a PCIe DMA read request, the following sequence of actions occurs:

1. After the dma_start signal asserts, the tx_dma_rd block in the TX application logic starts the DMA read transaction. It generates the DMA read request(s) based on the information obtained from the DMA registers and loads them into the TX command FIFO buffers.

2. After the TX PCIe packets cross the clock domain boundary in the TX command FIFO buffers, the tx_pcie block reads them in.
3. If the PCI Express MegaCore function is ready to accept the read completion packet, the tx_pcie block transmits the generated TLP and asserts the tx_st_sop and tx_st_valid signals. If the PCI Express MegaCore function is not ready to accept the memory read request, or if there are no transmit credits available for the TX read requests, the TX read request detours to the read bypass FIFO. This way, the tx_pcie block can then send the subsequent non-read request instead.

4. As the last DW of header or data is sent, tx_pcie asserts the tx_st_eop signal.

5. After the PCI Express MegaCore function accepts the DMA read request, it reformats it and sends it onto the PCIe link.

6. After some time, the PCIe link returns with a read completion request.

7. The PCI Express MegaCore function asserts rx_st_sop and H0 and H1 of the header is available on rx_st_data. The RX application logic looks at the format (Fmt) and type (Type) fields. The transaction type can be determined to be a read completion request.

8. The rx_sop signal is de-asserted and H2, containing the address information, is available on rx_st_data.

9. The application logic extracts the necessary information, reformats the request, and puts the request into the RX command FIFO.

10. The data enters and accumulates in the RX Data FIFO.

11. After crossing the clock domain boundary, the rx_ddr block reads in the RX read completion request.

12. When enough data is present, the rx_ddr block generates the DDR read commands and sends the data to the DDR2 SDRAM controller.

13. The DDR2 SDRAM controller forwards the data with a write command to the DDR2 SDRAM memory to store the data.

**Endpoint-Initiated DMA Write**

To initiate a DMA write request, the Root Port first programs all the DMA registers sequentially. After the DMA control register is written and decoded, the dma_start signal asserts for one clock cycle to initiate the DMA write.

When processing a PCIe DMA write request, the following sequence of actions occurs:

1. When the RX application logic block (rx_pcie) detects the dma_start signal, it generates the RX PCIe packet(s) based on the information in the DMA registers. The rx_pcie block writes the RX PCIe packet(s) into the RX command FIFO.

2. In the DDR2 clock domain, the rx_ddr block accepts the read request from the command FIFO and translates it into one or more DDR2 read requests. The rx_ddr block sends the read requests to the DDR2 SDRAM controller sequentially.

3. The DDR2 SDRAM controller reformats the read requests and sends them to the DDR2 SDRAM memory.

4. After some time, the DDR2 SDRAM memory returns the read request data.
5. The data accumulates in the TX data FIFO. At the same time, the `tx_ddr_resp` block in the TX application logic monitors this action and determines when to send the read data to the PCI Express MegaCore function.

6. When enough data accumulates in the TX data FIFO, the `tx_ddr_resp` block in the TX application logic generates a PCIe write packet and puts it in the TX command FIFO.

7. When enough data accumulates in the TX data FIFO, `tx_ddr_resp` generates a PCIe memory write packet and puts it in the TX command FIFO.

8. Through the TX data FIFO, the packet crosses the clock domain and the `tx_pcie` logic reads it as a PCIe TX packet.

9. If the PCI Express MegaCore function is ready to accept the read completion packet, `tx_pcie` transmits the generated TLP, and asserts the `tx_st_sop` and `tx_st_valid` signals.

10. As the last DW of header or data is sent, `tx_pcie` asserts the `tx_st_eop` signal.

**Memory Mapping**

In this reference design, the PCI Express MegaCore function reserves 16 MBytes of system address space (refer to Table 1) accessed through two base address registers `BAR0` and `BAR1`, and 4 KBytes of system address space for DMA registers through `BAR2`.

<table>
<thead>
<tr>
<th>Table 1. Memory Address Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Region</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>BAR0 and BAR1</td>
</tr>
<tr>
<td>BAR2</td>
</tr>
</tbody>
</table>

**Note to Table 1:**

*(1)* Although the PCIe address has a 32- or 64-bit width, the application logic only presents 24 bits to the DDR2 SDRAM controller.

BAR0 and BAR1 specify the DDR2 memory address range. The PCI Express function claims a target transaction if there is a BAR match with `BAR0` and/or `BAR1` based on 32-bit or 64-bit addressing, as specified in the request descriptor.

BAR2 maps the DMA registers (refer to Table 2).

<table>
<thead>
<tr>
<th>Table 2. Internal Registers Memory Mapped Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range Reserved</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>0x00000 - 0x00003</td>
</tr>
<tr>
<td>0x00004 - 0x00007</td>
</tr>
<tr>
<td>0x00008 - 0x0000B</td>
</tr>
<tr>
<td>0x0000C - 0x0000F</td>
</tr>
<tr>
<td>0x00010 - 0x00013</td>
</tr>
</tbody>
</table>
DMA Registers

This section describes the DMA registers of the DMA engine. This reference design implements the following registers:

- PCIe Address Register (PAR)
- DMA Byte Counter Register (BCR)
- Control Status Register (CSR)
- Local Address Register (LAR)

These registers, memory-mapped to BAR2 of the PCIe function, allow any logic module in the FPGA or any component that interfaces to the FPGA to configure their content prior to initiating a transfer.

To activate the DMA registers, the DMA requires writing to the DMA registers in the following sequence:

1. Write the PAR with the PCIe starting address for the current DMA transaction.
2. Write the BCR with the number of bytes to transfer.
3. Write the LAR with the DDR2 SDRAM memory starting address of the transaction.
4. Write the CSR with the appropriate value.

Writing a logical 1 into bit 7 of the CSR triggers the DMA state machine. The next clock cycle sets the dma_busy bit of the CSR register to indicate that the DMA transfer is in progress. The DMA engine sends the requests to the corresponding block to initiate the DMA transfer. The receiver block transfers the data between the DDR2 SDRAM memory and the system memory. The DMA engine de-asserts the dma_busy signal when the DMA transfer completes and a new DMA transaction may start.

**PCIe Address Register (PAR)**

The PCIe address is 64 bits long but the PAR is a 32-bit register. Therefore, the PCIe address register comprises two 32-bit registers. The PCIe upper address register (HPAR) and the PCIe lower address register (LPAR). Combining the HPAR and the LPAR forms the PCIe address. For 32-bit PCIe request addressing, only the LPAR bits are valid. For 64-bit PCIe request addressing, the combined 64-bit address is valid.

The PAR contains the PCIe bus address for the current DMA transfer and stays unchanged until reprogrammed. The PCIe bus DMA transfer initiated by the DMA engine must begin at the QWORD boundary. Table 3 shows the PAR format.

**Table 3. DMA PCI Address Register Format**

<table>
<thead>
<tr>
<th>Data Bit</th>
<th>Name</th>
<th>Read/Write</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>31..0</td>
<td>PAR</td>
<td>Write</td>
<td>32-bit address</td>
</tr>
</tbody>
</table>
DMA Byte Counter Register (BCR)
The DMA byte counter register is a 13-bit register. A 13-bit counter implements the BCR. The BCR holds the byte count for the current DMA memory transfer. A PCIe bus DMA transfer initiated by the DMA engine requires a QWORD transfer. Table 4 shows the BCR format.

Table 4. DMA Byte Counter Register Format

<table>
<thead>
<tr>
<th>Data Bit</th>
<th>Name</th>
<th>Read/Write</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>21..0</td>
<td>BCR</td>
<td>Write</td>
<td>13-bit counter</td>
</tr>
<tr>
<td>31..13</td>
<td>—</td>
<td>—</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Control Status Register (CSR)
The 32-bit CSR configures the DMA engine. The CSR directs the DMA operation and provides the status of the current DMA transfer. Table 5 describes the CSR format.

Table 5. DMA Byte Counter Register Format

<table>
<thead>
<tr>
<th>Data Bit</th>
<th>Mnemonic</th>
<th>Read/Write</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>5..0</td>
<td>—</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>6</td>
<td>Read/Write</td>
<td>Write</td>
<td>DMA readwrite. Asserted high for a DMA write. De-asserted for a DMA read.</td>
</tr>
<tr>
<td>7</td>
<td>DMA Start</td>
<td>Write</td>
<td>DMA start. Asserted high indicates all DMA registers are programmed and the DMA transaction is ready to begin.</td>
</tr>
<tr>
<td>30..8</td>
<td>—</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>31</td>
<td>DMA Busy</td>
<td>Read</td>
<td>DMA busy. Asserted high indicates the current DMA is still in progress and the DMA registers are not ready for new programming.</td>
</tr>
</tbody>
</table>

Local Address Register (LAR)
The 32-bit LAR holds the SDRAM address from which the data transfers to and from the SDRAM. Implemented with a 23-bit counter, the LAR is a write-only register. Table 6 shows the format of the LAR.

Table 6. DMA Local Address Register Format

<table>
<thead>
<tr>
<th>Data Bit</th>
<th>Name</th>
<th>Read/Write</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>22..0</td>
<td>LAR</td>
<td>Write</td>
<td>23-bit address</td>
</tr>
<tr>
<td>31..23</td>
<td>—</td>
<td>—</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Interface Signal Descriptions

This section describes the signals of the reference design application logic in two categories:

- Signals Interfacing with the PCI Express MegaCore Function
- Signals Interfacing with the DDR2 SDRAM Controller
For information on the local signals of the PCI Express MegaCore function, refer to the *PCI Express Compiler User Guide*. For information on the local signals of the DDR2 SDRAM High-Performance Controller, refer to the *DDR & DDR2 SDRAM High-Performance Controller v9.0 User Guide*.

## Signals Interfacing with the PCI Express MegaCore

Table 7 describes the data signals that interfaces with the PCI Express MegaCore function from `applogic.v`.

<table>
<thead>
<tr>
<th>Signal</th>
<th>I/O</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rstn_i</td>
<td>Input</td>
<td>Reset signal for the application logic. The signal is active low.</td>
</tr>
<tr>
<td>PCIeClk_i</td>
<td>Input</td>
<td>The 250 MHz clock fed by <code>coreclk_out</code> of the PCI Express MegaCore.</td>
</tr>
<tr>
<td>Rx_St_Data_i[63:0]</td>
<td>Input</td>
<td>Receive streaming data signals from the PCI Express MegaCore. The signal is 64 bits wide and all TLPs are received on this signal. For more information about this signal, refer to the <em>PCI Express Compiler User Guide</em>.</td>
</tr>
<tr>
<td>Rx_St_Valid_i</td>
<td>Input</td>
<td>Receive streaming data valid signal from the PCI Express MegaCore. This signal is asserted when logic clocks valid data on the <code>Rx_St_Data_i</code> signal. For more information about this signal, refer to the <em>PCI Express Compiler User Guide</em>.</td>
</tr>
<tr>
<td>Rx_St_Sop_i</td>
<td>Input</td>
<td>Receive streaming SOP signal from the PCI Express MegaCore. This signal is asserted for one clock cycle at the start of a TLP transmission on <code>Rx_St_Data_i</code>. For more information about this signal, refer to the <em>PCI Express Compiler User Guide</em>.</td>
</tr>
<tr>
<td>Rx_St_Eop_i</td>
<td>Input</td>
<td>Receive streaming EOP signal from the PCI Express MegaCore. This signal is asserted for one clock cycle at the end of a TLP transmission on <code>Rx_St_Data_i</code>. For more information about this signal, refer to the <em>PCI Express Compiler User Guide</em>.</td>
</tr>
<tr>
<td>Rx_St_Bardec_i[7:0]</td>
<td>Input</td>
<td>Receive streaming BAR decode signal from the PCI Express MegaCore. This signal states that BAR is being targeted for the TLP transmission on <code>Rx_St_Data_i</code>. This design can have BAR0 or BAR2 targeted. For more information about this signal, refer to the <em>PCI Express Compiler User Guide</em>.</td>
</tr>
<tr>
<td>Rx_St_Be_i[7:0]</td>
<td>Input</td>
<td>Receive streaming byte enable signal from the PCI Express MegaCore. Each bit references the corresponding byte on <code>Rx_St_Data_i</code>. The corresponding bit is high if the byte is valid. For more information about this signal, refer to the <em>PCI Express Compiler User Guide</em>.</td>
</tr>
<tr>
<td>Rx_St_Ready_o</td>
<td>Output</td>
<td>Receive streaming ready signal from the PCI Express MegaCore. This signal is asserted when the application is ready to accept data. For more information about this signal, refer to the <em>PCI Express Compiler User Guide</em>.</td>
</tr>
</tbody>
</table>
## Table 7. Signals Interfacing with the PCI Express MegaCore (Part 2 of 2)

<table>
<thead>
<tr>
<th>Signal</th>
<th>I/O</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rx_St_Mask_o</td>
<td>Output</td>
<td>Receive streaming mask signal from the PCI Express MegaCore. This signal is asserted when the application is not ready to accept non-posted requests. For more information about this signal, refer to the <a href="#">PCI Express Compiler User Guide</a>.</td>
</tr>
<tr>
<td>Tx_St_Data_o[63:0]</td>
<td>Output</td>
<td>Transmit streaming data signals to the PCI Express MegaCore. This signal is 64 bits wide and all TLPs are sent on this signal. For more information about this signal, refer to the <a href="#">PCI Express Compiler User Guide</a>.</td>
</tr>
<tr>
<td>Tx_St_Valid_o</td>
<td>Output</td>
<td>Transmit streaming data valid signal to the PCI Express MegaCore. This signal is asserted when TLP data is valid on the Tx_St_Data_i signal. For more information about this signal, refer to the <a href="#">PCI Express Compiler User Guide</a>.</td>
</tr>
<tr>
<td>Tx_St_Sop_o</td>
<td>Output</td>
<td>Transmit streaming SOP signal to the PCI Express MegaCore. This signal is asserted for one clock cycle at the start of a TLP transmission on Tx_St_Data_i. For more information about this signal, refer to the <a href="#">PCI Express Compiler User Guide</a>.</td>
</tr>
<tr>
<td>Tx_St_Eop_o</td>
<td>Output</td>
<td>Transmit streaming EOP signal to the PCI Express MegaCore. This signal is asserted for one clock cycle at the end of a TLP transmission on Tx_St_Data_i. For more information about this signal, refer to the <a href="#">PCI Express Compiler User Guide</a>.</td>
</tr>
<tr>
<td>Tx_St_Ready_i</td>
<td>Input</td>
<td>Transmit streaming ready signal from the PCI Express MegaCore. This signal is asserted when the PCI Express MegaCore is ready to receive data. For more information about this signal, refer to the <a href="#">PCI Express Compiler User Guide</a>.</td>
</tr>
<tr>
<td>TxCred_i[35:0]</td>
<td>Input</td>
<td>Transmit streaming credit signal from the PCI Express MegaCore. This signal describes which types of transactions the PCI Express MegaCore is able to accept. For more information about this signal, refer to the <a href="#">PCI Express Compiler User Guide</a>.</td>
</tr>
<tr>
<td>DevCsr_i[31:0]</td>
<td>Input</td>
<td>From the configuration module used to configure the PCI Express MegaCore. States the maximum read size.</td>
</tr>
<tr>
<td>BusDev_i[12:0]</td>
<td>Input</td>
<td>From the configuration module used to configure the PCI Express MegaCore. Used to determine the completer and requester IDs.</td>
</tr>
</tbody>
</table>

## Signals Interfacing with the DDR2 SDRAM High-Performance Controller

Table 8 describes the data signals that interface with the DDR2 SDRAM High-Performance Controller. All signals interface with rx_top or tx_top. For simplicity, the names of the signals correspond with the naming convention of the DDR2 SDRAM High-Performance Controller.
<table>
<thead>
<tr>
<th>Signal</th>
<th>I/O</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>local_ready</td>
<td>Input</td>
<td>Input to rx_top (DdrReady_i). This signal is asserted when the controller is ready to accept requests. For more information, refer to the DDR &amp; DDR2 SDRAM High-Performance Controller v9.0 User Guide.</td>
</tr>
<tr>
<td>local_address[24:0]</td>
<td>Output</td>
<td>Output from rx_top (DdrAddress_o). This signal holds the address of the memory that is to be written to or read from. For more information, refer to the DDR &amp; DDR2 SDRAM High-Performance Controller v9.0 User Guide.</td>
</tr>
<tr>
<td>local_write_req</td>
<td>Output</td>
<td>Output from rx_top (DdrWrite_o). This signal is asserted when a write is requested to the address on local_address. For more information, refer to the DDR &amp; DDR2 SDRAM High-Performance Controller v9.0 User Guide.</td>
</tr>
<tr>
<td>local_read_req</td>
<td>Output</td>
<td>Output from rx_top (DdrRead_o). This signal is asserted when a read is requested from the address on local_address. For more information, refer to the DDR &amp; DDR2 SDRAM High-Performance Controller v9.0 User Guide.</td>
</tr>
<tr>
<td>local_wdata_req</td>
<td>Output</td>
<td>Input to rx_top (DdrWrDataReq_i). This signal is asserted by the controller when the controller is ready to accept data on the next clock cycle. For more information, refer to the DDR &amp; DDR2 SDRAM High-Performance Controller v9.0 User Guide.</td>
</tr>
<tr>
<td>local_wdata[127:0]</td>
<td>Output</td>
<td>Output from rx_top (DdrWriteData_o). This signal contains the data to be written to the memory. Although the data is 128 bits wide, only 64 bits are used. This is because the specific memory component chosen is 128 bits wide. For more information, refer to the DDR &amp; DDR2 SDRAM High-Performance Controller v9.0 User Guide.</td>
</tr>
<tr>
<td>local_be</td>
<td>Output</td>
<td>Output from rx_top (DdrByteEnable_o). This signal is a byte enable signal that masks off individual bytes of local_wdata during writes. For more information, refer to the DDR &amp; DDR2 SDRAM High-Performance Controller v9.0 User Guide.</td>
</tr>
<tr>
<td>local_size</td>
<td>Output</td>
<td>Output from rx_top (DdrBurstCount_o). This signal is the burst size of the requested access, encoded as a binary number. For more information, refer to the DDR &amp; DDR2 SDRAM High-Performance Controller v9.0 User Guide.</td>
</tr>
<tr>
<td>local_rdata[63:0]</td>
<td>Input</td>
<td>Input to tx_top (TxReadData_i). This signal contains data returned from the memory after a read request is initiated from local_read_req. For more information, refer to the DDR &amp; DDR2 SDRAM High-Performance Controller v9.0 User Guide.</td>
</tr>
<tr>
<td>local_rdata_valid</td>
<td>Input</td>
<td>Input to tx_top (TxReadDataValid_i). This signal is asserted when the data on local_rdata is valid. For more information, refer to the DDR &amp; DDR2 SDRAM High-Performance Controller v9.0 User Guide.</td>
</tr>
</tbody>
</table>
Simulating the Reference Design

The following sections describe how to simulate the reference design. The Quartus II software generates a testbench model when you use the MegaWizard Plug-In Manager to generate a PCI Express core. This design incorporates that simulation model. The testbench is for use with the ModelSim-Altera edition.

Simulation Design Hierarchy

In the files generated from the Quartus II Software, the PCI Express core (pex.v) is instantiated in pex_example_chaining_pipen1b.v. This design replaces the instantiation of the PCI Express core with the top-level file (pcie_ddr2.v). For simplicity, the memory model generated for the DDR2 SDRAM High-Performance Controller is referenced within the Endpoint Example Design (pex_example_chaining_pipen1b.v) (refer to Figure 20).

Figure 20. Top-Level Quartus II Generated Testbench

All of the files in the testbench were generated by the Quartus II software when the PCI Express MegaCore was generated, with the exception of the following:

- pcie_ddr2.v—top level design file
- applogic.v and all files referenced within—application logic including the instantiation of the DDR2 SDRAM High-Performance Memory Controller
- ddr2_ctrl_hp_mem_model.v—generated by the Quartus II software when the DDR2 SDRAM High-Performance Memory Controller is generated
If the PCI Express MegaCore is regenerated, the following files are overwritten and must be modified:

- **pex_example_chaining_pipen1b.v**—instantiates the top-level design file (pcie_ddr2.v) and the memory module. If the ports on the PCI Express MegaCore change, the **pcie_ddr2.v** file also must be changed.

- **altpcietb_bfm_driver_chaining.v**—chaining DMA test driver file containing the commands the Root Complex sends.

Use the chaining DMA test driver to provide inputs to the serial side of the PCI Express core. A number of instructions are available, as referenced in Chapter 7 of the *PCI Express Compiler User Guide*.

The chaining DMA test driver provides instructions to the Root Port and the instructions are translated into a serial format that theoretically feed the rx_in ports of the PCI Express core. To simplify the simulation, the instructions are sent in parallel through the pipe interconnection module (PIPE).

We use the PCI Express MegaCore simulation model generated by the Quartus II software because it conforms to the *PCI Express Base Specification, Rev 2.0*. The complexity of the configuration of the core and the interaction within the physical, data link and transaction layers are better handled by the model generated by the Quartus II software.

### Chaining DMA Test Driver Commands

Use the **ebfm_barwr** and **ebfm_barrd** procedures. These procedures are shown in Table 9 and Table 10.

**Table 9. ebfm_barwr Procedure**

<table>
<thead>
<tr>
<th>Location</th>
<th>altpcietb_bfm_rdwr.v or altpcietb_bfm_rdwr.vhd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax:</td>
<td>ebfm_barwr (bar_table, bar_num, pcie_offset, lcladdr, byte_len, tclass)</td>
</tr>
<tr>
<td>Arguments:</td>
<td></td>
</tr>
<tr>
<td>bar_table</td>
<td>Address of the Endpoint bar_table structure in BFM shared memory. The bar_table structure stores the address assigned to each BAR so that the driver code does not have to be aware of the actual assigned addresses, only the application-specific offsets from the BAR.</td>
</tr>
<tr>
<td>bar_num</td>
<td>Number of the BAR used with pcie_off set to determine PCI Express address.</td>
</tr>
<tr>
<td>pcie_offset</td>
<td>Address offset from the BAR base.</td>
</tr>
<tr>
<td>lcladdr</td>
<td>BFM-shared memory address of the data to be written.</td>
</tr>
<tr>
<td>byte_len</td>
<td>Length, in bytes, of the data written. Can be 1 to the minimum of the bytes remaining in the BAR space or BFM shared memory.</td>
</tr>
<tr>
<td>tclass</td>
<td>Traffic class used for the PCI Express transaction.</td>
</tr>
</tbody>
</table>

**Table 10. ebfm_barrd Procedure (Part 1 of 2)**

<table>
<thead>
<tr>
<th>Location</th>
<th>altpcietb_bfm_rdwr.v or altpcietb_bfm_rdwr.vhd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax:</td>
<td>ebfm_barrd_wait (bar_table, bar_num, pcie_offset, lcladdr, byte_len, tclass)</td>
</tr>
</tbody>
</table>

© April 2009 Altera Corporation  AN 575: PCI Express-to-DDR2 SDRAM Reference Design
Before using these procedures, you have to temporarily write the data into the Root Port BFM and then specify where you wrote the memory to with the lcladdr arguments. **Figure 21** shows the Root Port address space.

**Figure 21.** Root port Address Space

<table>
<thead>
<tr>
<th>Addr</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 0000</td>
<td>Root Complex Shared Memory</td>
</tr>
<tr>
<td>0x001F FF80</td>
<td>Configuration Scratch Space Used by BFM routines. Not writable by user calls or endpoint.</td>
</tr>
<tr>
<td>0x001F FFC0</td>
<td>BAR Table Used by BFM routines. Not writable by user calls or endpoint.</td>
</tr>
<tr>
<td>0x0020 0000</td>
<td>Endpoint I/O Space BARs assigned smallest to largest.</td>
</tr>
<tr>
<td>0xFFFF FFFF</td>
<td>Unused</td>
</tr>
</tbody>
</table>

The address space from 0x0000 0000 to 0x001F FF80 is available to store data that can be transmitted using the procedures. The value of the address is specified by the lcladdr argument. The length of the data is specified by the byte_len argument.
To write data to the Root Port address space, use the `shmem_fill` procedure (refer to Table 11). The `addr` argument corresponds to the `lcladdr` argument in the read and write procedures described previously.

**Table 11. shmem_fill Procedure**

<table>
<thead>
<tr>
<th>Location:</th>
<th>altpcietb_bfm_shmem.v or altpcietb_bfm_shmem.vhd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax:</td>
<td><code>shmem_fill (addr, mode, leng, init)</code></td>
</tr>
<tr>
<td>Arguments:</td>
<td><code>addr</code> BFM shared memory starting address for filling data.</td>
</tr>
<tr>
<td></td>
<td><code>mode</code> Data pattern used for filling the data.</td>
</tr>
<tr>
<td></td>
<td><code>leng</code> Length, in bytes, of data to fill. If the length is not a multiple of the incrementing data pattern width, the last data pattern is truncated to fit.</td>
</tr>
<tr>
<td></td>
<td><code>init</code> Initial data value used for incrementing data pattern modes in VHDL. This argument is type <code>Std_logic_vector [63 down to 0]</code>. In Verilog HDL, this argument is <code>reg[63:0]</code>. In both languages, the necessary least significant bits are used for the data patterns that are smaller than 64 bits.</td>
</tr>
</tbody>
</table>

The mode argument of the `shmem_fill` procedure is specified by the constants defined in Table 12.

**Table 12. Mode Arguments for the shmem_fill Procedure**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHMEM_FILL_ZEROS</td>
<td>Specifies a data pattern of all zeros.</td>
</tr>
<tr>
<td>SHMEM_FILL_BYTE_INC</td>
<td>Specifies a data pattern of incrementing 8-bit bytes (0×00, 0×01, 0×02, etc.)</td>
</tr>
<tr>
<td>SHMEM_FILL_WORD_INC</td>
<td>Specifies a data pattern of incrementing 16-bit words (0×0000, 0×0001, 0×0002, etc.)</td>
</tr>
<tr>
<td>SHMEM_FILL_DWORD_INC</td>
<td>Specifies a data pattern of incrementing 32-bit dwords (0×0000000, 0×0000001, 0×0000002, etc.)</td>
</tr>
<tr>
<td>SHMEM_FILL_QWORD_INC</td>
<td>Specifies a data pattern of incrementing 64-bit qwords (0×000000000, 0×000000001, 0×000000002, etc.)</td>
</tr>
<tr>
<td>SHMEM_FILL_ONE</td>
<td>Specifies a data pattern of all ones</td>
</tr>
</tbody>
</table>

**Generating Testbench Stimulus**

In the main procedure, near the bottom of the `altpcietb_bfm_driver_chaining.v` file in the `/pex_examples/chaining_dma/testbench` directory, the following commands are used.

To first configure the PCI Express core, use the `ebfm_cfg_rp_ep` procedure in Example 1.

**Example 1.**

```vhdl
  ebfm_cfg_rp_ep(
    bar_table,         // BAR Size/Address info for Endpoint
    1,                 // Bus Number for Endpoint Under Test
    1,                 // Device Number for Endpoint Under Test
    512,               // Maximum Read Request Size for Root Port
    1,                 // Display EP Config Space after setup
    addr_map_4GB_limit // Limit the BAR assignments to 4GB address map
  );
```
To specify the data to be written to the FPGA, the `shmem_fill` command writes the data into the 0x0 address space of the virtual Root Complex (refer to Example 2).

**Example 2.**

```
// Shared Memory Fill: Write data to Root Complex memory to hold
// Memory      Data Pattern          Length of          Starting Data
// Address              Type      Bytes to Fill          Pattern Value
shmem_fill( 0, SHMEM_FILL_DWORD_INC,           64, 4’hAAAAAA00BBBBBB00);
```

For both the DDR2 SDRAM memory (BAR0 and BAR1) and the DMA Registers (BAR2), consider `pcie_offset` to be 0. The testbench generates the addresses for the Endpoint by referencing an offset that is added to the addresses specified in the `ebfm` procedures (refer to Example 3).

**Example 3.**

```
pcie_offset = 32’h0;
```

The `ebfm_barwr` write procedure now sends a memory write TLP containing the data that is held in the address space of the virtual Root Complex to the DDR2 SDRAM memory (refer to Example 4).

**Example 4.**

```
// ebfm_barwr(bar_table, bar_num, pcie_offset, lcladdr, byte_len, tclass)
ebfm_barwr(bar_table, 0, pcie_offset, 0, 64, 0);
```

The `ebfm_barrd` read procedure now sends a memory read TLP requesting the data that was just written to the DDR2 SDRAM memory (refer to Example 5).

**Example 5.**

```
// ebfm_barrd_wait(bar_table, bar_num, pcie_offset, lcladdr, byte_len, tclass)
ebfm_barrd_wait(bar_table, 0, pcie_offset, 32’hF0, 64, 0);
```

**Running the Simulation**

The simulation takes approximately 382,000 ns to run. This is because both the PCI Express hard IP and the DDR2 SDRAM High-Performance Controller IP have simulation models that require initialization, which takes quite a while. To perform the simulation, follow these steps.

**Restore Archived Project and Open Project**

The `.qar` includes all of the files for a Quartus II compile and a Modelsim simulation.

1. Select **Project**, then **Restore Archived Project**.
2. Browse to **pcie_ddr2.qar**.
3. Click **OK** to confirm.
4. Click **Yes** to create new directory.
5. Ensure **Open restored project** is checked.
6. Click **OK** to confirm.
Compile Project
This step compiles the top-level design file `pcie_ddr2.v`.
1. Click to compile the project.
2. View the compilation results.

Simulate Design in ModelSim
Perform these steps to simulate the design with the ModelSim software. It is not necessary to compile the design in the Quartus II software to simulate in ModelSim.
1. Open ModelSim-Altera.
2. Select File, then Change Directory…
3. Browse to `<top level directory>`\`pex_examples\chaing_dma\testbench`
4. Click OK.
5. At the Modelsim> command prompt, type `do runtb.do`.

View Simulation Results
The Root Complex sends a MemWr TLP to the Endpoint writing data to the DDR2 SDRAM memory (refer to Figure 22).

![Figure 22. MemWr TLP](image)

The Root Complex then sends a MemRd TLP to the Endpoint requesting the data (refer to Figure 23).

![Figure 23. MemRd TLP](image)

The Endpoint then returns a CplD TLP with the data to the Root Complex (refer to Figure 24).

![Figure 24. CplD TLP](image)
Reference Documents

This application note referenced the following documents:

- PCI Express Compiler User Guide
- DDR & DDR2 SDRAM High-Performance Controller v9.0 User Guide

Document Revision History

Table 13 shows the revision history for this application note.

<table>
<thead>
<tr>
<th>Date and Document Version</th>
<th>Changes Made</th>
<th>Summary of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>April 2009, v1.0</td>
<td>Initial release.</td>
<td>—</td>
</tr>
</tbody>
</table>