# Table of Contents

Table of Contents ................................................................................................................. 2
Table of Tables ......................................................................................................................... 3
Table of Figures ......................................................................................................................... 4

1 Overview ................................................................................................................................. 6
  1.1 Feature List ......................................................................................................................... 6
  1.2 Supported Products ............................................................................................................. 6
  1.3 DMA Design Overview ..................................................................................................... 7
  1.4 Software/Driver Overview ................................................................................................. 8
  1.5 Board Architecture Overview .......................................................................................... 9
    1.5.1 ‘PCIe Noconfig’ Netlist ............................................................................................... 9
    1.5.2 ConfigFPGA Designs ................................................................................................. 10

2 Software Interface ................................................................................................................. 12
  2.1 DMA Design Internal Operation ....................................................................................... 14
  3.1 BAR Accesses .................................................................................................................... 14
    3.1.1 BAR0 Accesses ........................................................................................................... 14
    3.1.2 Userspace BAR Accesses (BAR1-4) ......................................................................... 19
    3.1.3 Timeouts ..................................................................................................................... 20
    3.1.4 Transaction Ordering ................................................................................................. 21
  3.2 DMA Accesses .................................................................................................................. 22
    3.2.1 Host-Memory Descriptor Mode ................................................................................... 22
    3.2.2 FPGA-Memory Descriptor (Demand-mode) Accesses ............................................... 24
    3.2.3 Transaction Ordering ................................................................................................. 26

4 DMA Design Interface Operation .......................................................................................... 27
  4.1 Data Alignment ................................................................................................................ 27
  4.2 ‘DiniBus’ Interface .......................................................................................................... 28
    4.2.1 FromHost Transactions ............................................................................................. 32
    4.2.2 ToHost Transactions ................................................................................................. 32
  4.3 AXI Interface .................................................................................................................... 32
  4.4 ‘Split’ Interface ................................................................................................................ 33
    4.4.1 Target FromHost Transactions .................................................................................. 38
    4.4.2 Target ToHost Transaction ...................................................................................... 38
    4.4.3 Target ToHost Transaction Gotchas .......................................................................... 39
    4.4.4 Host-Memory FromHost DMA .................................................................................. 39
    4.4.5 Host-Memory ToHost DMA ...................................................................................... 41
    4.4.6 Host-Memory DMA Gotchas ..................................................................................... 42
    4.4.7 Demand-Mode ToHost DMA ..................................................................................... 43
    4.4.8 Demand-Mode FromHost DMA ................................................................................ 44
    4.4.9 Demand-Mode Gotchas ............................................................................................. 45

5 ‘Combined’ Interface ............................................................................................................ 46
  4.5.1 FromHost Write Transactions ..................................................................................... 49
  4.5.2 ToHost Read Transactions .......................................................................................... 49

5 Interrupts ................................................................................................................................. 50
5.1 DMA Interrupts ............................................................................................................ 50
5.2 User Interrupts ............................................................................................................ 50
5.3 Software ....................................................................................................................... 50
5.4 MSI Message Vectors ................................................................................................. 51
6 Performance .................................................................................................................... 54
   6.1 Bandwidth .................................................................................................................. 54
   6.2 Latency ....................................................................................................................... 55
   6.3 Optimal Software Strategy ........................................................................................ 56
      6.3.1 Host-Mode DMA ................................................................................................. 56
      6.3.2 Demand-Mode DMA .......................................................................................... 57
Appendix A: Simulation Testbench .................................................................................... 58
      Files for configFPGA ..................................................................................................... 58
      Files for User FPGA ..................................................................................................... 58
      PCIe Simulation Model ............................................................................................... 59
      Modeling Software Interaction .................................................................................. 60
Appendix B: Example User Design .................................................................................. 61
Appendix C: Indeterminate-Size ToHost DMA ................................................................. 67
Appendix D: Card-to-Card DMA ....................................................................................... 68
Appendix E: Troubleshooting/FAQ .................................................................................. 69
      Device fails to enumerate .............................................................................................. 69
      Timeouts on Read Data ................................................................................................. 69
      DMA Hang .................................................................................................................... 69
      Driver Fails to Compile/Install .................................................................................... 69
Contact Info ....................................................................................................................... 70

Table of Tables
Table 1 – Change Log ........................................................................................................... 5
Table 2 – BAR0 Address Map ............................................................................................. 14
Table 3 – Host-Memory Descriptor Description ................................................................ 23
Table 4 – Demand Mode 2-Qword Descriptor Format ....................................................... 25
Table 5 – DiniBus Signal Definitions .................................................................................. 28
Table 6 – Split Interface Parameter Definitions .................................................................. 33
Table 7 – Split IF User Signals ............................................................................................ 33
Table 8 – Combined Interface Parameter Definitions ....................................................... 46
Table 9 – Combined Interface Signal Definitions .............................................................. 47
Table 10 - Example User Design Parameters .................................................................... 62
Table 11 - Example User Design Register Address Map ................................................... 62
# Table of Figures

- Figure 1 – PCIe Gen3 DMA Design Block Diagram ................................................................. 7
- Figure 2 - PCIe Noconfig Block Diagram ............................................................................. 9
- Figure 3 - ConfigFPGA Design Block Diagram ................................................................. 10
- Figure 4 – Address Translation from Host PC Memory to User Design Address .................. 19
- Figure 5 - Host-Memory Descriptor Format ......................................................................... 22
- Figure 6 - Demand-mode Descriptor Format ....................................................................... 24
- Figure 7 - Data Offsets ......................................................................................................... 27
- Figure 8 - DiniBus FromHost Transaction Waveform .......................................................... 32
- Figure 9 - DiniBus ToHost Transaction Waveform ............................................................ 32
- Figure 10 - Split IF FromHost Target Transaction ................................................................ 38
- Figure 11 - Split IF ToHost Target Transaction ................................................................... 38
- Figure 12 – Split IF Host-Memory FromHost DMA Beginning ............................................. 39
- Figure 13 – Split IF Host-Memory FromHost DMA Throttling ............................................. 40
- Figure 14 – Split IF Host-Memory FromHost DMA Ending ............................................... 40
- Figure 15 - Split IF Host-Memory ToHost DMA Beginning ................................................. 41
- Figure 16 – Split IF Host-Memory ToHost DMA Valid ....................................................... 41
- Figure 17 - Split IF Host-Memory ToHost DMA Throttling ............................................... 42
- Figure 18 - Split IF Host-Memory ToHost DMA Ending ..................................................... 42
- Figure 19 - Split IF Demand-Mode ToHost DMA Beginning .............................................. 43
- Figure 20 - Split IF Demand-Mode ToHost DMA Valid ..................................................... 43
- Figure 21 - Split IF Demand-Mode ToHost DMA Throttling .............................................. 43
- Figure 22 - Split IF Demand-Mode Tohost DMA Ending .................................................... 44
- Figure 23 - Split IF Demand-Mode FromHost DMA Beginning ........................................ 44
- Figure 24 - Split IF Demand-Mode FromHost DMA Throttling ......................................... 45
- Figure 25 - Split IF Demand-Mode FromHost DMA Ending ............................................. 45
- Figure 26 – Combined IF FromHost Write ......................................................................... 49
- Figure 27 – Combined IF ToHost Read ............................................................................... 49
- Figure 28 - Performance Testing Design Block Diagram ................................................... 54
- Figure 29 - User Design Block Diagram ............................................................................. 61
<table>
<thead>
<tr>
<th>Revision</th>
<th>Name</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Brian Poladian</td>
<td>Initial</td>
</tr>
<tr>
<td>1.2</td>
<td>Brian Poladian</td>
<td>Added max_payload and max_read_req BAR0 registers</td>
</tr>
<tr>
<td>1.3</td>
<td>Brian Poladian</td>
<td>Added timeout value for invalid addresses. Added timeout limit parameter to interface modules.</td>
</tr>
<tr>
<td>1.4</td>
<td>Brian Poladian</td>
<td>Added MSI vectors</td>
</tr>
<tr>
<td>1.5</td>
<td>Brian Poladian</td>
<td>Added TLP debug registers</td>
</tr>
<tr>
<td>1.6</td>
<td>Brian Poladian</td>
<td>Fixed TLP error register decode</td>
</tr>
<tr>
<td>1.7</td>
<td>Brian Poladian</td>
<td>Added AXI interface. Updated split interface address cycle data.</td>
</tr>
<tr>
<td>1.8</td>
<td>Brian Poladian</td>
<td>Fix for TSK_TARGET_WRITE arguments. Updated board list.</td>
</tr>
</tbody>
</table>
1 Overview

1.1 Feature List
- Three independent, full-duplex, high performance DMA engines
- Full software and driver support in both Windows and Linux
- Source code provided for software and driver free-of-charge
- Simple address/data or FIFO interfaces in RTL
- Fully backwards compatible with previous DiniGroup designs
- Scalable width and speed interfaces

1.2 Supported Products
The Dini PCIe Gen3 DMA design will support all boards with a Xilinx Ultrascale FPGA on them. At the time of writing, this list includes:
- DNPCIE_40G_KU_LL
- DNPCIE_40G_KU_LL_2QSFP
- DNPCIE_80G_A10_LL
- DNPCIE_400G_VU_LL
- DNPCIE_400G_VUP_LL
- DNVUF1A
- DNVUF2_PCIE
- DNVUF2_HPC_PCIE
- DNVUF2A
- DNVUF4A
1.3 DMA Design Overview

The DiniGroup PCIe Gen3 DMA design is intended to provide users with simple, out-of-the-box software and RTL interfaces for transferring data to and from a DiniGroup board. The design wraps the Xilinx Ultrascale FPGA Gen3 Integrated Block for PCI Express with interfaces for managing BAR and DMA accesses, which allows users to access RTL designs in FPGA using a single 64-bit address space. The design is provided as a netlist that can be integrated into any RTL design, and also comprises the heart of all configFPGA designs distributed on DiniGroup Ultrascale boards. The design is highly configurable and is capable of satisfying both high bandwidth requirements and low latency applications. Configurations can also be scaled for minimal FPGA resources.

Figure 1 – PCIe Gen3 DMA Design Block Diagram
1.4 Software/Driver Overview

The AETest software and driver package (available for download at http://www.dinigroup.com/files/web_packs/Aetest.zip) provides APIs for interacting with the DMA design. These APIs can be used directly in any C/C++ program or, for simple accesses or debugging, a text-based menu is available in the AETest executable. The EMU application (http://www.dinigroup.com/files/web_packs/emu.zip) uses the AETest APIs for all PCIe-based accesses.

The AETest software/driver function calls are responsible for managing BAR windows and DMA descriptor lists, and basic users do not need to understand the implementation details. However, advanced users who want to optimize performance should understand what’s going on under the hood. The basics of the board access functions are as follows:

**Target (BAR) Accesses**

1) At startup, map BARs into host PC memory space
2) For BAR0 accesses, perform the memory access and return (with data, if appropriate)
3) For BAR1-4 accesses, update the BAR0 registers so that the window into the 64-bit board address space is appropriate. Then perform the BAR 1-4 access and return (with data, if appropriate)

**DMA Accesses**

1) At startup, allocate DMA buffers
2) For each access, reserve a buffer
3) Add a descriptor into the descriptor list memory with a pointer to the buffer used for the transaction and settings for the transfer
4) Check descriptor list for access completion, return data to user if appropriate, and clear buffer
1.5 Board Architecture Overview
The PCIe Gen3 DMA design has applications in both a standalone user design and in the configFPGA infrastructure on larger prototyping systems. In a standalone design, the DMA design netlist can provide a direct, high bandwidth connection to a host PC; all DiniGroup ‘LL’ boards use the DMA design netlist as part of the Darklite package (contact support@dinigroup.com for more information on Darklite). In the configFPGA, the DMA design facilitates the movement of data from a host PC to all FPGAs on the board, as well as performing board control features through software-controllable configuration registers. Full register maps are included for all configFPGA designs in the board support package and will not be described in this document.

1.5.1 ‘PCIe Noconfig’ Netlist
The PCIe Gen3 DMA Design netlist is provided in board support packages at FPGA_Reference_Designs/licensed/pcie/pcie_noconfig_ultrascale.

Figure 2 - PCIe Noconfig Block Diagram

The netlist will be used directly in all ‘LL’ card reference designs, and users can also instantiate the netlist directly in a design on larger DiniGroup prototyping boards. The standard netlist contains three DMA engines and a single DiniBus interface, but netlists can be customized for fewer DMA engines or more DiniBus interfaces. For example, the Darklite design uses three DMA engines and three DiniBus interfaces.
Each DiniBus interface is assigned a different portion of the 64-bit board address space, and uses the upper 8 bits of the address space for partitioning. Downstream interface modules may further subdivide the address space into smaller regions, but the basic netlist has the following mapping:

- DiniBus IF 0: 0x00000000_00000000 – 0x00FFFFFF_FFFFFFFC
- DiniBus IF 1: 0x01000000_00000000 – 0x01FFFFFF_FFFFFFFC (if present)
- DiniBus IF 2: 0x02000000_00000000 – 0x02FFFFFF_FFFFFFFC (if present)

Using a netlist with multiple DiniBus interfaces is a simple way to connect multiple, independent streams of data to various design interfaces.

### 1.5.2 ConfigFPGA Designs

The PCIe Gen3 DMA Design is used in all configFPGA designs, which are distributed as bitfiles with DiniGroup prototyping boards. No source is provided for configFPGA designs, but full register maps are available in the board support package at FPGA_Reference_Designs/common/pcie/Documents.

The DMA design is attached to configuration registers, which control FPGA programming, clock frequencies, EEPROM access, misc I2C functions, etc., and also to TMB IO modules that allow data transfer to user FPGAs. The design also includes a “Hyperpipe” interface that allows communication between multiple instances of the DMA design, but that interface is beyond the
scope of this document and is used for internal software development only. The address map for the configFPGA is leverages the same type of address split as the Noconfig netlist. The upper 8 bits of the 64-bit address are used to determine the interface being accessed.

Hyperpipe: 0xFE000000_00000000 – 0xFEFFFFFF_FFFFFFFC
Config Registers: 0xFF000000_00000000 – 0xFFFFFFFF_FFFFFFFF
TMB IO to FPGA A: 0x00000000_00000000 – 0x00FFFFFF_FFFFFFFC
TMB IO to FPGA B: 0x01000000_00000000 – 0x01FFFFFF_FFFFFFFF (if present)
TMB IO to FPGA C: 0x02000000_00000000 – 0x02FFFFFF_FFFFFFFF (if present)
...

Like the Noconfig netlist, this addressing scheme allows the lower 56 bits of the address to be further subdivided within the user RTL design. In the standard MainRef design, the 56-bit address space is partitioned as follows:

Blockram: 0xXX000000_00000000 – 0xXX00FFFF_FFFFFFFF
Blockram2: 0xXX010000_00000000 – 0xXX01FFFF_FFFFFFFF
Memory: 0xXX020000_00000000 – 0xXX02FFFF_FFFFFFFF
Memory2: 0xXX030000_00000000 – 0xXX03FFFF_FFFFFFFF
Interconnect: 0xXX040000_00000000 – 0xXX04FFFF_FFFFFFFF
Registers: 0xXX080000_00000000 – 0xXX08FFFF_FFFFFFFF

See the file FPGA_Reference_Designs/common/pcie/pcie_dma/user_fpga_ultrascale/ dini_interface_combined_if_mainref_wrapper.v for partitioning details. The features of the MainRef design vary based on board features – consult the reference design included in the board support package for details.
2 Software Interface

The primary AETest function calls are as follows:

**Target (BAR) Accesses**

const char* **regwrite_boardspace_dword**(uint64_t board_address, uint32_t data)

Description: Writes a 32-bit dword into the 64-bit board address space using BAR1

Parameters:
- uint64_t board_address: 64-bit board address
- uint32_t data: 32-bit data

Return value: “success” if successful, otherwise an error string

const char* **regread_boardspace_dword**(uint64_t board_address, uint32_t* data)

Description: Reads a 32-bit dword from the 64-bit board address space using BAR1

Parameters:
- uint64_t board_address: 64-bit board address
- uint32_t* data: Pointer to 32-bit data read from the board

Return value: “success” if successful, otherwise an error string

**DMA Accesses**

const char* **dmawrite_boardspace_fromhost**(uint32_t dma_num, uint64_t board_address, uint32_t* data, uint32_t size_dwords)

Description: Writes buffer of dwords into the 64-bit board address space using DMA

Parameters:
- uint32_t dma_num: DMA engine to use. Valid values are 0, 1, 2.
- uint64_t board_address: 64-bit board address
- uint32_t* data: Pointer to buffer of 32-bit data dwords
- uint32_t size_dwords: Number of 32-bit data dwords in the data buffer

Return value: “success” if successful, otherwise an error string

const char* **dmaread_boardspace_tohost**(uint32_t dma_num, uint64_t board_address, uint32_t* data, uint32_t size_dwords, uint32_t* dwords_transferred)

Description: Reads buffer of dwords from the 64-bit board address space using DMA

Parameters:
- uint32_t dma_num: DMA engine to use. Valid values are 0, 1, 2.
- uint64_t board_address: 64-bit board address
- uint32_t* data: Pointer to buffer of 32-bit data dwords
- uint32_t size_dwords: Number of 32-bit data dwords to request be read into the data buffer
- uint32_t* dwords_transferred: Actual number of dwords transferred

Return value: “success” if successful, otherwise an error string

These APIs hide a considerable amount of implementation details that will be described in this document. When using the software APIs it’s not absolutely necessary to understand all of the
control mechanisms of the design, and for most users it will be acceptable to treat the DMA Design as a black box where software controls go in and the RTL interface comes out.

The above functions are all that’s needed for basic communication with the board, but for increased performance of DMA there is also an API for queueing non-blocking reads.

const char* dmaread_boardspace_tohost_nonblocking(uint32_t dma_num, uint64_t board_address, uint32_t* data, uint32_t size_dwords, void** waithandles)
Description: Queues read of dwords from the 64-bit board address space using DMA and returns immediately.
Parameters:
uint32_t dma_num: DMA engine to use. Valid values are 0, 1, 2.
uint64_t board_address: 64-bit board address
uint32_t * data: Pointer to buffer of 32-bit data dwords
uint32_t size_dwords: Number of 32-bit data dwords to request be read into the data buffer
void** waithandles: Tokens passed from driver that must be passed back in tohost_waitfinish call to identify the buffers used in the transfer
Return value: “success” if successful, otherwise an error string

const char* dmaread_boardspace_tohost_waitfinish(void** waithandles, unsigned char* got_eof, uint32_t timeout_seconds, uint32_t* dwords_transferred)
Description: Waits for completion of DMA and reads buffer of dwords from the 64-bit board address space
Parameters:
void** waithandles: Tokens passed from driver in nonblocking call to identify the buffers used in the transfer
unsigned char* got_eof: EOF bit set in descriptor
uint32_t timeout_seconds: Number of seconds to wait for DMA read to complete. Set to -1 for infinite wait, or 0 to check for completion and return immediately.
uint32_t * dwords_transferred: Actual number of dwords transferred
Return value: “success” if successful, otherwise an error string
# DMA Design Internal Operation

## 3.1 BAR Accesses

BAR0 is used for PCIe and BAR configuration, and BARs 1, 2, and 4 are windows into 64-bit board address space. The BAR windows can be moved independently and may overlap. The BAR windows are controlled by registers on the BAR0 interface.

### 3.1.1 BAR0 Accesses

**Table 2 – BAR0 Address Map**

<table>
<thead>
<tr>
<th>Byte Addr</th>
<th>Name</th>
<th>Bit Definitions</th>
<th>Detailed Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x004 R</td>
<td>Date</td>
<td>YYYY_MMDD</td>
<td>Date that this design was created/built. Example value is 0x2015_0518</td>
</tr>
<tr>
<td>0x008 R</td>
<td>Design Type</td>
<td>ASCII of Bitstream Type.</td>
<td>Possible values are: “Full” = 0x46756C6C “DARK” = 0x4441524B</td>
</tr>
<tr>
<td>0x00C RW</td>
<td>Reset Control</td>
<td>[0] BAR0 Reset (RW) [1] User Reset (RO)</td>
<td>In a configFPGA design, BAR0 Reset is a user-controllable active-high reset that will cause reset to be asserted to all modules that interface with the user design, and also all DMA state machines. User Reset is a read-only signal that is asserted either by PCIe reset or when internal clocks are not ready. User Reset will also reset all user interface and DMA modules.</td>
</tr>
<tr>
<td>0x010 RW</td>
<td>Reset Control</td>
<td>[0] BAR0 Reset [1] User Reset</td>
<td>Same control as 0x00C. Register added for V6/V5 register map compatibility.</td>
</tr>
<tr>
<td>0x018 RW</td>
<td>LED Control</td>
<td>[7:0] LED out [7:0] [15:8] LED en [7:0] [23:16] LED in [7:0]</td>
<td>Bits [15:0] are R/W Bits [23:16] are read only. LED order: {debug[2:0], yellow_activity, red_los, green_link8, green_link4, green_link1}</td>
</tr>
<tr>
<td>0x020 R/W</td>
<td>DMA0 Base Address0</td>
<td>[31:12] Base Address [11:0] 0x000</td>
<td>Lower 32 bit byte address of physical address (in host memory) where the DMA0 descriptor chain starts. This address must have the lower bytes cleared to match the DMA0 Address Mask register.</td>
</tr>
<tr>
<td>0x024 RW</td>
<td>DMA0 Base Address1</td>
<td>[31:0] Base Address</td>
<td>Upper 32 bits of Base Address[63:0], to form a 64 bit address. Set to 0 if using 32 bit addressing.</td>
</tr>
<tr>
<td>Address</td>
<td>Description</td>
<td>Details</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>0x028 RW</td>
<td>DMA0 Address Mask</td>
<td>[19:12] Address Mask [11:0] 0xFFF Address mask to indicate size of Descriptor list. Set some number of consecutive lower bits to 1. Set to the size of the list in bytes, minus 1. For example, for an 8KB descriptor list, set this to 8K-1=0x01FFF.</td>
<td></td>
</tr>
<tr>
<td>0x02C RW</td>
<td>DMA0 Control</td>
<td>[9:0] re-read time in milliseconds 10: DMA enable 11: DMA clear 12: DMA Demand Mode enable Re-read time is the time between checks for a valid bit in the next descriptor in the list. This value cannot be set to 0. If software tries to set this to zero, it will be set to one. Default value is 1. DMA enable enables the DMA engine. DMA clear clears the DMA engine, resetting the address counters to the base address and clearing any running transactions. Software must set this back to zero to bring the DMA engine out of “reset”. DMA Demand Mode enables the User FPGA to initiate to-host and from-host DMA transfers</td>
<td></td>
</tr>
<tr>
<td>0x030 W</td>
<td>DMA0 Poll Immediate</td>
<td>0: Poll Immediate Writing a 1 to this register causes the DMA engine to fetch the next descriptor from the host immediately (and not wait for the re-read timer to expire).</td>
<td></td>
</tr>
<tr>
<td>0x034 R</td>
<td>DMA0 Read Address</td>
<td>[31:0] Current Read Address DEBUG INFO ONLY: The lower 32 bits of the current address being read, or the next address that will be read.</td>
<td></td>
</tr>
<tr>
<td>0x038 R</td>
<td>DMA0 Execute Address</td>
<td>[31:0] Current Execute Address DEBUG INFO ONLY: The lower 32 bits of the address of the descriptor currently being worked on.</td>
<td></td>
</tr>
<tr>
<td>0x03C R</td>
<td>DMA0 Descriptor FIFO Info</td>
<td>[9:0] Descriptor FIFO empty count DEBUG INFO ONLY: Number of elements (qwords) in the descriptor FIFO. Note there could be more descriptors still pending in the system that have already been read out of this FIFO.</td>
<td></td>
</tr>
<tr>
<td>0x040-0x05C</td>
<td>Repeat for DMA1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x060-0x07C</td>
<td>Repeat for DMA2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x080 R</td>
<td>PCIe Clock Counter</td>
<td>[31:0] Clock Counter Counter for clock from the Xilinx PCIe Block.</td>
<td></td>
</tr>
<tr>
<td>0x084 R</td>
<td>Internal Clock Counter</td>
<td>[31:0] Clock Counter Counter for clock to internal DMA engines, TLP state machine, etc.</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>Type</td>
<td>Description</td>
<td>Details</td>
</tr>
<tr>
<td>---------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0x088</td>
<td>User Clock Counter</td>
<td>[31:0] Clock Counter</td>
<td>Counter for clocks to the user interfaces.</td>
</tr>
</tbody>
</table>
| 0x0D0   | Interrupt     | [0] DMA0 To Host Interrupt   
|          |               | [1] DMA0 From Host Interrupt   
|          |               | [2] DMA1 To Host Interrupt   
|          |               | [3] DMA1 From Host Interrupt   
|          |               | [4] DMA0 Going Idle   
|          |               | [5] DMA1 Going Idle   
|          |               | [6] DMA2 Going Idle   
|          |               | [7] DMA2 To Host Interrupt   
<p>|          |               | [8] DMA2 From Host Interrupt   | Bits will be asserted by the DMA engines when an interrupt condition occurs. Software needs to write a 1 to the corresponding bit to clear the interrupt. |
| 0xD4    | Interrupt Mask | [8:0] Mask Control for Interrupt Bits                                      | A value of 1 in the mask means that the corresponding interrupt source generates an interrupt over the PCI-E bus. A value of 0 means the corresponding interrupt bit doesn’t create an interrupt event. |
| 0xE0    | User Interrupt Passthrough | [31:0] Passthrough Control                                                  | A value of 1 means that the latch on User Interrupts is disabled, and the interrupt condition will clear as soon as the User Interrupt in the user FPGA is cleared. The default value of 0 enables the latch and is recommended to prevent spurious interrupts. |
| 0xE8    | User Interrupt | [31:0] User Interrupts                                                      | Bits will be asserted when an interrupt occurs in the user design. Software needs to write a 1 to the corresponding bit to clear the interrupt. |
| 0xEC    | User Interrupt Mask | [31:0] Mask Control for User Interrupt Bits                               | A value of 1 in the mask means that the corresponding interrupt source generates an interrupt over the PCI-E bus. A value of 0 means the corresponding interrupt bit doesn’t create an interrupt event. |</p>
<table>
<thead>
<tr>
<th>Offset</th>
<th>Address Name</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xF0 RW</td>
<td>BAR1 Upper Address</td>
<td>[31:0] Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Upper 32 bits of 64-bit target address for BAR1.</td>
</tr>
<tr>
<td>0xF4 RW</td>
<td>BAR1 Lower Address</td>
<td>[31:0] Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lower 32 bits of 64-bit target address for BAR1. Use mask register to control how many bits this register actually controls.</td>
</tr>
<tr>
<td>0xF8 RW</td>
<td>BAR2 Upper Address</td>
<td>[31:0] Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Upper 32 bits of 64-bit target address for BAR2.</td>
</tr>
<tr>
<td>0xFC RW</td>
<td>BAR2 Lower Address</td>
<td>[31:0] Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lower 32 bits of 64-bit target address for BAR2. Use mask register to control how many bits this register actually controls.</td>
</tr>
<tr>
<td>0x100 RW</td>
<td>BAR4 Upper Address</td>
<td>[31:0] Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Upper 32 bits of 64-bit target address for BAR4.</td>
</tr>
<tr>
<td>0x104 RW</td>
<td>BAR4 Lower Address</td>
<td>[31:0] Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lower 32 bits of 64-bit target address for BAR4. Use mask register to control how many bits this register actually controls.</td>
</tr>
<tr>
<td>0x108 RW</td>
<td>BAR1 Lower Address Mask</td>
<td>[31:0] Mask</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sets number of bits that lower 32-bit address register controls. Defaults to 0xFFC0_0000.</td>
</tr>
<tr>
<td>0x10C RW</td>
<td>BAR2 Lower Address Mask</td>
<td>[31:0] Mask</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sets number of bits that lower 32-bit address register controls. Defaults to 0xFF80_0000.</td>
</tr>
<tr>
<td>0x110 RW</td>
<td>BAR4 Lower Address Mask</td>
<td>[31:0] Mask</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sets number of bits that lower 32-bit address register controls. Defaults to 0xFF00_0000.</td>
</tr>
<tr>
<td>0x120 RW</td>
<td>BAR Timeout Value</td>
<td>[31:0] Timeout Value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Number of cycles to wait for a read response to be generated by user code before returning an 0xDEADBEE2_DEADBEE3 timeout value. Defaults to 0x00000FFF (4K clock cycles).</td>
</tr>
<tr>
<td>Offset</td>
<td>Feature</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 0x138 RW | PCIe Payload Size        | [2:0] Max payload used  
[17:15] Max payload available from Endpoint                                                                                                                                   | Maximum PCIe payload used can be less than or equal to the max payload available (reported by Endpoint). Values are:  
3’h0: 128 bytes  
3’h1: 256 bytes  
3’h2: 512 bytes  
3’h3: 1024 bytes  
3’h4: 2048 bytes  
3’h5: 4096 bytes  
3’h7: 64 bytes |
| 0x13C RW | PCIe Read Request Size   | [2:0] Max read request used  
[17:14] Max read request available from Endpoint                                                                                                                                  | Maximum PCIe read request used can be less than or equal to the max read request available (reported by Endpoint). Values are:  
3’h0: 128 bytes  
3’h1: 256 bytes  
3’h2: 512 bytes  
3’h3: 1024 bytes  
3’h4: 2048 bytes  
3’h5: 4096 bytes  
3’h7: 64 bytes |
| 0x170 R  | PCIe TLP Error           | [7:0] TLP Error Code  
[15:8] TLP Tag  
[31:16] TLP Lower Address                                                                                                                                                    | Reports any TLP errors reported by the PCIe endpoint, along with the tag and address for that TLP. Error codes are:  
0x1: Completion TLP is Poisoned  
0x2: Request terminated by a Completion w/ UR/CA/CRS status  
0x3: Request terminated by a Completion with no data/incorrect byte count  
0x4: Completion ID/TC/ATTR mismatch  
0x5: Completion starting address mismatch  
0x6: Invalid TAG  
0x7: Invalid byte count  
0x8: Request terminated by a Function-Level Reset  
0x9: Request terminated by a Completion Timeout |
3.1.2 Userspace BAR Accesses (BAR1-4)
BAR accesses into the 64-bit board address space provide a good way to access registers or monitor the progress of an algorithm.

If using the AETest API function calls, the BAR1-4 windows are moved automatically to the desired portion of the 64-bit board address. However, moving the window does require extra BAR0 accesses, which may be undesirable for performance reasons – users may instead choose to access BAR1-4 directly, in which case the window would need to be manually moved.

The 64-bit board address space (address[63:0]) is comprised of three parts:
[63:32] : set by BAR0 HIADDR register
[31:size_of_bar] : set by BAR0 LOADDR register
[size_of_bar-1:0] : set by target address

The size_of_bar variable is set by the LOADDR_ MASK register, and by default uses the maximum BAR size. The 64-bit board address is assembled as follows:
\{HIADDR, ((LOADDR & LOADDR_MASK) | (offset & ~LOADDR_MASK))\}.

**Figure 4 – Address Translation from Host PC Memory to User Design Address**

<table>
<thead>
<tr>
<th>Host PC Memory</th>
<th>Dini DMA Design</th>
<th>User Design</th>
</tr>
</thead>
</table>
| BAR0
0xd1c00000     | Host_memory_addr + offset  | BAR0        |
| BAR1
0xd1800000     | Host_memory_addr + offset  | \{(BAR1_HIADDR, (BAR1_LOADDR & BAR1_LOADDR_MASK) + offset)\} |
| BAR2
0xd1000000     | Host_memory_addr + offset  | \{(BAR2_HIADDR, (BAR2_LOADDR & BAR2_LOADDR_MASK) + offset)\} |
| BAR4
0xd0000000     | Host_memory_addr + offset  | \{(BAR4_HIADDR, (BAR4_LOADDR & BAR4_LOADDR_MASK) + offset)\} |
Example:
Consider the above figure. The physical address of BAR0 in host memory is 0xd1c00000. Suppose we want to write to board address 0x01000000_02005678.

1) Map host memory address for BAR0 in host software
2) Write to BAR0 host_memory_address + 0x100 (BAR4_HIADDR register) the value of the upper 32 bits of the address, 0x01000000
3) Write to BAR0 host_memory_address + 0x104 (BAR4_LOADDR register) the value of the lower 32 bits of the address that fall outside the maximum size of the BAR. In this case, BAR4 size is 16MB, and so 0x02XXXXXX needs to be written. (By default, BAR4’s LOADDR_MASK is set to 0xFF000000).
4) The BAR4 window is now set up, and is pointing to 0x01000000_02000000. Map host memory address for BAR4 in host software.
5) Write to BAR4 host_memory_address + 0x5678 the desired value for board address 0x01000000_02005678.

The target interface supports byte, word (2-byte), dword (4-byte), and qword (8-byte) accesses; dword accesses are commonly generated by most host machine chipsets. Up to 32-byte transfers are supported, but not recommended. Accesses must be aligned to the size of the access, and may not cross a 256-bit boundary.

3.1.3 Timeouts
Special error code are generated by the PCIe core and returned to the host software in place of valid data when a target transaction times out. These are returned in order to maintain a valid PCIe link with the host PC.
0xDEADBEE0_DEADBEE1 : read access to invalid memory space
0xDEADBEE2_DEADBEE3 : read access to valid memory space, but read timed out

If 0xffffffff_ffffffff is returned, the PCIe link itself is failing, indicating that either bad (malformed) data is being sent from the user design to the PCIe core, or data is being returned in response to a transaction that has already timed out. Most systems will hang soon after this condition is hit.

If 0xDEADBEE2_DEADBEE3 is being returned but the user design is correctly returning data, the timeout cycle count may be too low. Generally, target reads should return immediately because the host PC CPU will stall while the read is pending, and it’s not a good idea to stall the CPU for extended periods of time. Therefore it is assumed that most designs that do not return data immediately after a read request have some other (protocol) problem; if this is not the case and the user logic simply takes a long but finite time to return data, the timeout value BAR0 register can be given a larger value.
3.1.4 Transaction Ordering
BAR read requests will not pass write transactions, and can be used to flush BAR writes to the device. Because BAR accesses stall the processor and can time out, BAR transactions will be given priority over DMA transactions and will bypass both ToHost and FromHost data. Thus, there is no guaranteed transaction ordering between BAR accesses and DMA accesses.
3.2 DMA Accesses
DMA accesses are used for high-bandwidth data transfer, or for data transfers that originate within the user design. For easy-to-debug operation, descriptors can be stored and managed in host memory, and software can initiate all data transfers. Latency-sensitive applications should have the card initiate data transfers and keep all descriptor information in the board’s address space.

3.2.1 Host-Memory Descriptor Mode
In host-memory descriptor mode, data transfers are initiated by software as follows:
1) Software will reserve a buffer in PC host memory
2) Software will copy data into that buffer from PC application space if needed
3) Software will write a descriptor in PC host memory space that has the physical address of the buffer for the transfer, the address within the board’s 64-bit address space for the access, the length, and settings for the transfer.
4) Optionally, the software will send a BAR0 write signaling that the DMA engine should read the descriptor list immediately. Otherwise, the DMA engine will automatically read the descriptor list on a 1ms timer.

After setup, the card will read the descriptor from the descriptor list and process the transfer. After the transfer is complete, the card will mark the descriptor as invalid. The software can then perform the following actions:
1) If data has been written into the host PC data buffer, copy that data into PC application space.
2) Mark the data buffer as available.

Because the board address space is 64 bits but only 32 bits of board address are available in a single descriptor, an “address update” descriptor can be issued to set the upper 32 bits of the desired 64-bit address space, and the address provided in the data transfer descriptor will set the lower 32 bits of the 64-bit address. The AETest PCIe driver will automatically issue “address update” descriptors as needed, and handle all buffer allocation and management.

![Figure 5- Host-Memory Descriptor Format](image-url)
<table>
<thead>
<tr>
<th>Dword Number</th>
<th>Bit Number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31</td>
<td>Valid</td>
<td>Bit indicating if the descriptor is valid. Software sets this to 1 when the descriptor is ready to be processed (Other dwords of the descriptor written, memory pointed to by PCI Address available), Hardware clears this to 0 when the descriptor has been processed completely. Software may not modify ANYTHING in the descriptor once the valid bit has been set to 1.</td>
</tr>
</tbody>
</table>
| 0            | 30         | Direction                 | 0 = From_Host (Move data from PCI Address to Board Address)  
1 = To_Host (Move data from Board Address to PCI Address)                                                                                     |
| 0            | 29         | Generate Interrupt        | Set to 1 to set High Bits of Board Address. Does not transfer any data. Used to access more than 4 GB of memory space on the user side. This sets distinct registers depending on the value of the Direction Bit (To_host and From_host can have different high bits of the board address). |
| 0            | 28         | Set High Address          | If 1, hardware will interrupt software once this descriptor has been processed completely. Bit is ignored when Set Registers Type is 1.                                                                        |
| 0            | 27         | RESERVED                  | Reserved/unused/undefined. Set to 0 for future backward compatibility.                                                                                                                                 |
| 0            | 26         | Use Byte Enables          | When set, bits [23:16] contain the first and last byte enables for the DMA transfer. Otherwise, these bits will contain the upper bits of dword length.                                                     |
| 0            | 25         | RESERVED                  | Reserved/unused/undefined. Set to 0 for future backward compatibility.                                                                                                                                 |
| 0            | 24         | Transfer Complete         | Status bit (set by Hardware). To_Host direction only, indicates user FPGA indicated this is end of the data for this transfer.                                                                           |
| 0            | 23:20      | First Dword Byte Enables  | If bit [26] is set, this contains the byte enables for the first dword of data in the transfer. Otherwise contains the upper bits of the dword length.                                                   |
|              | 19:16      | Last Dword Byte Enables   | If bit [26] is set, this contains the byte enables for the last dword of data in the transfer. Otherwise contains the upper bits of the dword length.                                                           |
| 0            | 15:0       | Dword Length              | Length of the transfer measured in 32-bit (4-byte) dwords. This length must indicate a contiguous block of memory in PCI address space and user address space.                                             |
| 1            | 31:0       | Board Address             | Byte Address to read or write from on the board (user side address space). When bit [28] is asserted, this is the upper 32 bits of a 64-bit board byte address.                                      |
| 2            | 31:0       | PCI Address               | Physical address to read or write data in host memory. Note that the data must be address contiguous for this entire transfer.                                                                              |
| 3            | 31:0       | PCI Address               | Upper 32 bits of PCI physical Address. If software is using 32 bit addressing, this must be set to 0.                                                                                                       |
3.2.2 FPGA-Memory Descriptor (Demand-mode) Accesses

All three DMA engines support Demand Mode transfers initiated by user design. In this mode, descriptors come from user design and are not stored in host memory as in the “normal” mode. Each DMA engine can only support one of the two modes; the mode is set by writing to a BAR0 register (defaults to “normal” mode), and will remain in demand-mode until the register is changed back. Demand-mode accesses are not supported on the “Combined IF” module.

The user design initiates a demand mode transfer as follows:
1) Write a 2-qword descriptor with host PC buffer address, board address, size, and control to the dma_to_host port.
2) If transferring data to the host, the user design will write data on the dma_to_host port following the descriptor.

The DMA design will respond to the user as follows
3) A 2-qword descriptor will be returned on the dma_from_host port signaling that the transfer was processed.
4) If a from-host transfer was requested, the DMA design will write data on the dma_from_host port following the descriptor.

As with the host-memory-mode DMA, a 32-bit board address is allowed in each data descriptor, which will set the lower 32 bits of the 64-bit board address space. To set the upper 32 bits of the 64-bit address, an “address update” descriptor can be sent. No completion responses are issued on the dma_from_host port in response to “address update” descriptors. The 64-bit board address will determine the address that completion descriptors are returned to, so it’s essential that the address is correct.

Figure 6 - Demand-mode Descriptor Format
<table>
<thead>
<tr>
<th>Qword Number</th>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63:32</td>
<td>Return Board Address</td>
<td>The return address used either for the write completion descriptor or the read return data. When bit 28 is 0, this will be the lower 32 bits of the return address, and when bit 28 is 1, this will be the upper 32 bits of the return address. All return data will be aligned according to this address. The return address can be treated instead as a tag for identifying return data, and in this case it is recommended to keep return_address[2:0]=3'b000 so that no realignment of data occurs.</td>
</tr>
<tr>
<td>0</td>
<td>31</td>
<td>Valid</td>
<td>Must be 1</td>
</tr>
<tr>
<td>0</td>
<td>30</td>
<td>Direction</td>
<td>0 = From_Host (Move data from PCI Address to User FPGA) 1 = To_Host (Move data from FPGA to PCI Address)</td>
</tr>
<tr>
<td>0</td>
<td>29</td>
<td>Generate Interrupt</td>
<td>If 1, hardware will interrupt software once this descriptor has been processed completely.</td>
</tr>
<tr>
<td>0</td>
<td>28</td>
<td>Set High Address</td>
<td>Set to 1 to set High Bits of Return Board Address. Does not transfer any data when this bit is set. Used to access more than 4 GB of memory space on the user side. This sets distinct registers depending on the value of the Direction Bit (To_host and From_host can have different high bits of the board address).</td>
</tr>
<tr>
<td>0</td>
<td>27</td>
<td>Reserved</td>
<td>Must set to 0</td>
</tr>
<tr>
<td>0</td>
<td>26</td>
<td>Dword Byte enables</td>
<td>Set to make this transaction a to-host transfer with byte enables, the enables are in bits [23:16]</td>
</tr>
<tr>
<td>0</td>
<td>25:24</td>
<td>Reserved</td>
<td>Must be 0</td>
</tr>
<tr>
<td>0</td>
<td>23:0</td>
<td>Dword size / byte enables</td>
<td>Number of dwords to transfer. This length must indicate a contiguous block of memory in PCI address space and user address space. If this is a byte enables transaction (Bit 26 is set) bits [23:20] are the byte enables for the first (or only) dword, bits [19:16] are byte enables for the last dword, they must not be 0. The number of dwords to transfer is in bits [15:0]. If this is not a byte enables transaction (Bit 26 is not set), [23:0] is the number of dwords to transfer.</td>
</tr>
<tr>
<td>1</td>
<td>63:0</td>
<td>PCI Address[63:0]</td>
<td>PCI address for the transfer. If software is using 32 bit addressing, bits [63:32] must be 0.</td>
</tr>
</tbody>
</table>

If using a data width larger than 64 bits, all bits above 64 will be ignored for the two cycles of descriptor data.

The AETest PCIe driver does not have built-in support for demand-mode accesses, but there are calls to allocate and reserve buffers. The PC application will need to manage the buffers directly and update the RTL user design with the physical buffer addresses to use in the demand-mode transfers.

3.2.3 Transaction Ordering
DMA engines operate independently, and there is no transaction order between DMA engines. All DMA operations are processed in the order that they are received from a descriptor list or from a demand-mode user design. Thus, read transactions can be used to flush write transactions for a single engine, but not between engines.
4 DMA Design Interface Operation

4.1 Data Alignment

The user RTL interfaces have the option to run at a maximum data width of 256 bits, but can be configured to run in 64 or 128-bit widths for reduced FPGA utilization (and also reduced bandwidth). The data provided to the user always begins aligned to address modulo the data width, which will mean that the data width parameter will directly affect how data is presented. For example, consider the following data alignments:

**Figure 7 - Data Offsets**

<table>
<thead>
<tr>
<th>Offset 0x0</th>
<th>127</th>
<th>63</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data[31:0]</td>
<td>Data[31:0]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset 0x4</th>
<th>127</th>
<th>63</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data[31:0]</td>
<td>Data[31:0]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset 0x8</th>
<th>127</th>
<th>63</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data[31:0]</td>
<td>Data[31:0]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset 0xC</th>
<th>127</th>
<th>63</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data[31:0]</td>
<td>Data[31:0]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset 0x10</th>
<th>127</th>
<th>63</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data[31:0]</td>
<td>Data[31:0]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset 0x14</th>
<th>127</th>
<th>63</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data[31:0]</td>
<td>Data[31:0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data[95:64]</td>
<td>Data[63:32]</td>
<td>Data[31:0]</td>
<td>0xXXXXXXXX</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset 0x18</th>
<th>127</th>
<th>63</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data[31:0]</td>
<td>Data[31:0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data[63:32]</td>
<td>Data[31:0]</td>
<td>0xXXXXXXXX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset 0x1C</th>
<th>127</th>
<th>63</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data[31:0]</td>
<td>Data[31:0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data[31:0]</td>
<td>0xXXXXXXXX</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For the highest data efficiency, it is recommended to use an address that is aligned to the data width. This prevents cycles of data transfer with lost payload due to address alignment.
4.2 ‘DiniBus’ Interface
The DiniBus interface is used internally in the DMA design and is also the interface exposed on the Noconfig netlist. It is generally not recommended to use this interface directly, but to instead use one of the Combined or Separate Interface modules as the interface to the user design because the DiniBus interface may be expanded or changed to meet new design requirements, whereas the other interface signals are guaranteed to keep their functionality for compatibility purposes. The DiniBus Interface is a fixed width of 256 bits (unlike the other configurable-width interfaces) and internally in the DMA design it runs at 250MHz. The DiniBus interface may run at a reduced frequency after being passed through a TMB link.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pcie_clk</td>
<td>FromHost</td>
<td>Clock for DiniBus signals listed below.</td>
</tr>
<tr>
<td>pcie_reset</td>
<td>FromHost</td>
<td>Synchronous reset for DiniBus signals listed below.</td>
</tr>
</tbody>
</table>
if (pcie_fromhost_valid==2'b10):
    [63:0] : 64-bit board address
    [127:64] : 64-bit host address
    [159:128] : length in dwords
    [163:160] : first dword byte enable
    [167:164] : last dword byte enable
    [168] : use dword byte enables
    [187:169] : buffer ID
    [193:192] : SOF/EOF
        2'h1: Normal control cycle, data follows if write
        2'h2: EOF control cycle, no data follows
        2'h3: Last control cycle of descriptor, data follows if write
    [194] : interrupt
    [195] : demand mode
    [199:196] : one-hot BAR selection
    [196] : BAR0
    [197] : BAR1
    [198] : BAR2
    [199] : BAR4
    [200] : Read transaction (0 for writes)
    [202:201] : Transaction type
        2'h0: Target transaction
        2'h1: DMA0 transaction
        2'h2: DMA1 transaction
        2'h3: DMA2 transaction
    [255:216] : Tag

if (pcie_fromhost_valid==2'b01):
    Data payload

FromHot
pcie_fromhost_transaction_type [3:0]
One-hot encoded transaction type
    4'h1: Target transaction
    4'h2: DMA0 transaction
    4'h4: DMA1 transaction
    4'h8: DMA2 transaction

FromHot
pcie_fromhost_valid [1:0]
One-hot control/payload selection
    2'h0 : pcie_fromhost_data is invalid
    2'h1 : pcie_fromhost_data is payload
    2'h2 : pcie_fromhost_data is control
| pcie_fromhost_almost_full [3:0] | FromHost | One-hot encoded almost full  
| | | 4’h1: Target almost full  
| | | 4’h2: DMA0 almost full  
| | | 4’h4: DMA1 almost full  
| | | 4’h8: DMA2 almost full  
| pcie_tohost_data [255:0] | ToHost | if (pcie_tohost_valid==2'b10) :  
| | | [63:0] : 64-bit board address  
| | | [127:64] : 64-bit host address  
| | | [159:128] : length in dwords  
| | | [163:160] : first dword byte enable  
| | | [167:164] : last dword byte enable  
| | | [168] : use dword byte enables  
| | | [187:169] : buffer ID  
| | | [192] : start of transaction  
| | | [193] : end of transaction  
| | | [194] : interrupt  
| | | [195] : demand mode  
| | | [199:196] : one-hot BAR selection  
| | | [196] : BAR0  
| | | [197] : BAR1  
| | | [198] : BAR2  
| | | [199] : BAR4  
| | | [200] : Read transaction (0 for writes)  
| | | [202:201] : Transaction type  
| | | 2’h0: Target transaction  
| | | 2’h1: DMA0 transaction  
| | | 2’h2: DMA1 transaction  
| | | 2’h3: DMA2 transaction  
| | | [255:216] : Tag  
| | if (pcie_tohost_valid==2'b01) :  
| | Data payload  
| pcie_tohost_transaction_type [3:0] | ToHost | One-hot encoded transaction type  
| | | 4’h1: Target transaction  
| | | 4’h2: DMA0 transaction  
| | | 4’h4: DMA1 transaction  
| | | 4’h8: DMA2 transaction  
| pcie_tohost_valid [1:0] | ToHost | One-hot control/payload selection  
| | | 2’h0 : pcie_tohost_data is invalid  
| | | 2’h1 : pcie_tohost_data is payload  
| | | 2’h2 : pcie_tohost_data is control  

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
</table>
| pcie_tohost_almost_full [3:0] | One-hot encoded almost full  
   4'h1: Target almost full  
   4'h2: DMA0 almost full  
   4'h4: DMA1 almost full  
   4'h8: DMA2 almost full |
| pcie_tohost_wr_count [39:0]   | Optional signal showing FIFO write counts for target and DMA interfaces.  
   [9:0] : Target FIFO write count  
   [19:10] : DMA0 FIFO write count  
   [29:20] : DMA1 FIFO write count  
   [39:30] : DMA2 FIFO write count |
### 4.2.1 FromHost Transactions

Figure 8 - DiniBus FromHost Transaction Waveform

All FromHost transactions consist of a single control word followed optionally by data. The ToHost almost full signal is used to throttle FromHost data. Read requests will not have data cycles.

### 4.2.2 ToHost Transactions

Figure 9 - DiniBus ToHost Transaction Waveform

All ToHost transactions consist of a single control word followed optionally by data. The FromHost almost full signal is used to throttle ToHost data. Read requests and EOF updates will not have data cycles.

### 4.3 AXI Interface

For users wanting to interface the DMA design directly to third-party IP, or for use with HLS, a 256-bit AXI streaming interface is available at common/pcie/pcie_dma/user_fpga_ultrascale/dini_interface_axi_if.v. The module attaches to a DiniBus interface from the PCIe netlist or TMB connection. The AXI streaming data will be identical to the DiniBus data, with control data always occurring on the first cycle of a packet optionally followed by payload data. Standard backpressure is available via the TREADY/TVALID signaling described in Xilinx’s [documentation](#).
4.4 ‘Split’ Interface

The ‘Split’ interface uses separate ports for target, DMA0, DMA1, and DMA2 transactions. Users may want to use this interface if the memory spaces accessed by each of the DMA engines are separate, if connecting DMA data directly to a FIFO where addresses would be ignored, or if Demand-mode DMA is required. In addition, this interface is compatible with that on the pcie_interface module in legacy V5/V6/V7 designs. The split interface Verilog module is located in the reference materials at common/pcie/pcie_dma/user_fpga_ultrascale/dini_interface_split_if.v. The module attaches to a DiniBus interface from the PCIe netlist or TMB connection; the DiniBus ports will not be described below – see DiniBus section for signal descriptions.

Table 6 – Split Interface Parameter Definitions

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA_WIDTH_IN</td>
<td>Width of internal DiniBus Interface data. Should always be set to 256.</td>
</tr>
<tr>
<td>DATA_WIDTH_OUT</td>
<td>Width of Split Interface data. Set to 64 for compatibility with legacy designs. Valid values are 64, 128, and 256.</td>
</tr>
<tr>
<td>TAG_WIDTH_OUT</td>
<td>Width of DMA tags. Will be 8 for 64-bit data, 10 for 128-bit data, and 14 for 256-bit data.</td>
</tr>
<tr>
<td>NUM_DMA_ENGINES</td>
<td>Number of DMA engines in the design. Should always be set to 3.</td>
</tr>
<tr>
<td>DMA_ENGINE_ENABLES</td>
<td>One-hot enable signal for DMA engines. Default to 3'b111 for all engines enabled.</td>
</tr>
<tr>
<td>ONECLOCK</td>
<td>Set to 1 if DiniBus and Split Interfaces run on the same clock (pcie_clk and user_clk are the same signal). Set to 0 otherwise (default).</td>
</tr>
<tr>
<td>TARGET_TIMEOUT_VALUE</td>
<td>Number of clock cycles to wait for a target read transaction to return before returning to an idle state. Value must be less than the BAR0 timeout register, defaults to 0x7FF (1K clock cycles).</td>
</tr>
</tbody>
</table>

Table 7 – Split IF User Signals

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>user_clk</td>
<td>FromHost</td>
<td>Clock for all Split Interface user signals listed below</td>
</tr>
<tr>
<td>user_reset</td>
<td>FromHost</td>
<td>Reset for all Split Interface user signals listed below, and user state machines/registers in Split IF module.</td>
</tr>
<tr>
<td>target_address [63:0]</td>
<td>FromHost</td>
<td>64-bit byte address for target transfers.</td>
</tr>
<tr>
<td>target_address_valid</td>
<td>FromHost</td>
<td>One clock cycle strobe that indicates the target_address is valid</td>
</tr>
<tr>
<td><strong>target_write_data</strong></td>
<td>FromHost</td>
<td>Target data.</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
<td>--------------</td>
</tr>
<tr>
<td>[DATA_WIDTH_OUT-1:0]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>target_write_be</strong></th>
<th>FromHost</th>
<th>Byte enables for target write data</th>
</tr>
</thead>
<tbody>
<tr>
<td>[DATA_WIDTH_OUT/8-1:0]</td>
<td></td>
<td>[0] : target_write_data[7:0] is valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...(similar pattern for all other bits)...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>target_write_enable</strong></th>
<th>FromHost</th>
<th>Indicates valid write data on this interface.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>target_write_accept</strong></th>
<th>ToHost</th>
<th>Accepts the valid write data. Data is allowed to transfer when target_write_accept and target_write_enable are active. If connecting to blockRAM, this signal can be tied high.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>debug_target_bar</strong></th>
<th>FromHost</th>
<th>Valid for reads and writes. Indicates which bar is being accessed (bar number 1, 2, or 4). To treat target_address as a unified 64-bit address space, this signals should be used for debug/informational purposes only.</th>
</tr>
</thead>
</table>
| [2:0]                 |          | 3’b001: BAR1 access  
|                       |          | 3’b010: BAR2 access  
|                       |          | 3’b100: BAR4 access |

<table>
<thead>
<tr>
<th><strong>target_read_enable</strong></th>
<th>FromHost</th>
<th>High on a target read request.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>target_request_tag</strong></th>
<th>FromHost</th>
<th>Tag associated with this read request.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[3:0]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>target_read_accept</strong></th>
<th>ToHost</th>
<th>Acceptance of read request. Allows interface to “move on” before the read data is returned by the user. Target_read_enable and target_read_accept being high signals “transfer” of request. If connecting to blockram, this signal can be tied high.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>target_read_data</strong></th>
<th>ToHost</th>
<th>Read Data to return to configFPGA.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[DATA_WIDTH_OUT-1:0]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>target_read_be</strong></th>
<th>FromHost</th>
<th>Byte enables for target read data</th>
</tr>
</thead>
<tbody>
<tr>
<td>[DATA_WIDTH_OUT/8-1:0]</td>
<td></td>
<td>[0] : target_read_data[7:0] is requested</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[1] : target_read_data[15:8] is requested</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...(similar pattern for all other bits)...</td>
</tr>
<tr>
<td><strong>target_read_data_tag</strong> [3:0]</td>
<td><strong>ToHost</strong></td>
<td>Tag that accompanies this data. This value must match the value provided on target_request_tag when target_read_enable was high.</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>target_read_data_valid</strong></td>
<td><strong>ToHost</strong></td>
<td>Clock cycle pulse indicating read data is valid.</td>
</tr>
<tr>
<td><strong>target_read_ctrl</strong> [7:0]</td>
<td><strong>FromHost</strong></td>
<td>Size of the target read requested in dwords – 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8’h0 : 1 dword (32 bits) requested</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8’h1 : 2 dword (64 bits) requested</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8’h2 : 3 dword (96 bits) requested</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8’h3 : 4 dword (128 bits) requested</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...(similar pattern for all other bits)...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Target read request size should always be less than or equal to the PCIe max payload size.</td>
</tr>
<tr>
<td><strong>target_read_data_ctrl</strong> [7:0]</td>
<td><strong>ToHost</strong></td>
<td>User must save target_read_ctrl[7:0] when target_read_enable is asserted and return it on target_read_data_ctrl[7:0]</td>
</tr>
<tr>
<td><strong>dma0_from_host_data</strong> [DATA_WIDTH_OUT-1:0]</td>
<td><strong>FromHost</strong></td>
<td>Address, length, or data, depending on the state of dma0_from_host_ctrl[5,0].</td>
</tr>
<tr>
<td><strong>dma0_from_host_ctrl</strong> [TAG_WIDTH_OUT-1:0]</td>
<td><strong>FromHost</strong></td>
<td>Host-Memory DMA:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{[5], [0]} indicates type of data.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2’b01: 64 bit board byte address.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2’b10: 24 bits of dword length [23:0]. First byte enable on [35:32] and last byte enable on [39:36]. Last transaction of transfer indicator on [40]. Upper 23 bits are reserved. For reads, user must return this many dwords of data, properly aligned based on bit 0 of the dword board address. For writes this information is provided but informational only. Read/write request bit is valid here.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2’b00: Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2’b11: Reserved/Undefined/Never Occurs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>([13:8], [3:2]) are dword enables</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[2] : dma0_from_host_data[31:0] valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[8] : dma0_from_host_data[95:64] valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[9] : dma0_from_host_data[127:96] valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[10] : dma0_from_host_data[159:128] valid</td>
</tr>
<tr>
<td>dma0_from_host_valid</td>
<td>FromHost</td>
<td>Indicates valid data on data and ctrl signals.</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>dma0_from_host_advance</td>
<td>ToHost</td>
<td>Indicates that the user design is ready to accept more data – the PCIe core will stop sending data when this signal is deasserted. This signal is intended to attach to an almost-full signal; dma0_from_host_valid may be asserted for several clock cycles after dma0_from_host_advance is deasserted due to pipelining delays.</td>
</tr>
<tr>
<td>dma0_to_host_data</td>
<td>ToHost</td>
<td>Address, length, or data, depending on the state of dma0_from_host_ctrl[4:2].</td>
</tr>
<tr>
<td>dma0_to_host_ctrl</td>
<td><strong>ToHost</strong></td>
<td><strong>Host-Memory DMA:</strong></td>
</tr>
<tr>
<td>-----------------</td>
<td>----------</td>
<td>---------------------</td>
</tr>
<tr>
<td>[TAG_WIDTH_OUT:0]</td>
<td></td>
<td>[0] : dma0_to_host_data[31:0] valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[2] : Demand Mode – set to 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[3] : Last data for this read request (EOF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[4] : Reserved – set to 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[5] : Reserved – set to 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[6] : Reserved – set to 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[7] : Reserved – set to 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[8] : dma0_to_host_data[95:64] valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[10] : dma0_to_host_data[159:128] valid</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th><strong>Demand-mode DMA:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] : dma0_to_host_data[31:0] valid</td>
<td>[0] : dma0_to_host_data[31:0] valid</td>
</tr>
<tr>
<td>[3] : Set with the last data on a to-host transfer, set on the 2\textsuperscript{nd} qword for the descriptor on a from-host transfer</td>
<td>[3] : Set with the last data on a to-host transfer, set on the 2\textsuperscript{nd} qword for the descriptor on a from-host transfer</td>
</tr>
<tr>
<td>[4] : Set during writing of the 2-qword descriptor for both to-host and from-host transfers. Set to 0 when transferring write (to-host) data</td>
<td>[4] : Set during writing of the 2-qword descriptor for both to-host and from-host transfers. Set to 0 when transferring write (to-host) data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dma0_to_host_valid</th>
<th><strong>ToHost</strong></th>
<th>Indicates valid data on data and ctrl signals.</th>
</tr>
</thead>
</table>

| dma0_to_host_almost_full| **FromHost**| When high, user should stop writing data to the module soon. |
4.4.1 Target FromHost Transactions

**Figure 10 - Split IF FromHost Target Transaction**

- target_write_accept must be asserted for the transaction to begin.
- target_address_valid will assert when target_address is valid.
- target_write_enable will assert when target_write_data and target_write_be are valid.

4.4.2 Target ToHost Transaction

**Figure 11 - Split IF ToHost Target Transaction**

- target_read_accept must be asserted for the transaction to begin.
- target_address_valid and target_read_enable will pulse when target_address, target_read_be, target_request_tag, and target_read_ctrl are valid.
- target_read_data_valid should be asserted when target_read_data, target_read_data_tag, and target_read_data_ctrl are valid.

### 4.4.3 Target ToHost Transaction Gotchas
- Target reads must complete within 4K clock cycles. The host processor will stall while the read is outstanding, and to prevent a permanent system hang, the configFPGA will return a timeout value after 4K clock cycles.
- The exact amount of data requested should be returned. Returning too much or too little data will result in unexpected/unsupported behavior.
- The common use for the target interface is a register interface that can always accept accesses; it is recommended that target_write_accept and target_read_accept always be asserted.
- Only one target read is issued at a time, and the host processor will hang while it waits for a response – it is encouraged to use this interface for low-latency accesses.
- target_request_tag and target_read_ctrl should be returned as target_read_request_tag and target_read_data_ctrl.

### 4.4.4 Host-Memory FromHost DMA

Figure 12 – Split IF Host-Memory FromHost DMA Beginning

<table>
<thead>
<tr>
<th>user_clk</th>
<th>dma_from_host_data[31:0]</th>
<th>dma_from_host_ctrl[7:0]</th>
<th>dma_from_host_valid</th>
<th>dma_from_host_advance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>address</td>
<td>length</td>
<td>data[0]</td>
<td>data[1]</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>0x0d</td>
<td>0x2c</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- dma_from_host_valid is asserted at start of transfer, and remains asserted while transfer data/control is active.
- 64-bit address is transferred on first valid cycle (dma_from_host_ctrl[5,0]=2’b01)
- If the data width of dma_from_host_data is greater than 64, all bits above bit 64 will be invalid/unused during the address cycle.
- dma_from_host_ctrl[4] should be 0 to indicate write.
- Length data is transferred on second valid cycle (dma_from_host_ctrl[5,0]=2’b10).
- If the data width of dma_from_host_data is greater than 64, all bits above bit 64 will be invalid/unused during the length cycle.
- Data is transferred from third valid cycle onward.
- `dma_from_host_advance` allows the write to continue; deasserting this signal prevents write data from advancing.
- `dma_from_host_valid` may be asserted for several clock cycles after `dma_from_host_advance` is deasserted due to pipelining delays.

- The transaction is considered officially over when the full length of data has been transferred. There is no last-cycle indicator.
- Note that the last cycle of data may have a different dword enable value (pictured above) depending on the length of data to be transferred.
4.4.5 Host-Memory ToHost DMA

**Figure 15 - Split IF Host-Memory ToHost DMA Beginning**

- `dma_from_host_valid` is asserted at start of transfer, and remains asserted while transfer data/control is active.
- 64-bit address is transferred on first valid cycle (`dma_from_host_ctrl[5,0]=2'b01`).
- If the data width of `dma_from_host_data` is greater than 64, all bits above bit 64 will be invalid/unused during the address cycle.
- `dma_from_host_ctrl[4]` should be 1 to indicate read.
- Length data is transferred on second valid cycle (`dma_from_host_ctrl[5,0]=2'b10`).
- If the data width of `dma_from_host_data` is greater than 64, all bits above bit 64 will be invalid/unused during the length cycle.
- Return data will begin sometime later – there is no latency requirement on reads.
- `dma_to_host_valid` will be asserted when read return data is valid.

**Figure 16 – Split IF Host-Memory ToHost DMA Valid**

- `dma_to_host_valid` should be deasserted when read return data is invalid; return data does not need to be continuous.
• If `dma_to_host_almost_full` is asserted, read return data should not advance and `dma_to_host_valid` should be deasserted.

**Figure 18 - Split IF Host-Memory ToHost DMA Ending**

- `dma_from_host_valid` is deasserted after last valid data.
- `dma_to_host_ctrl[3]` should be asserted on last valid data.

### 4.4.6 Host-Memory DMA Gotchas

- Allowable latencies for throttling are dependent on the depth of the FIFOs used in the design and the actual latencies are dependent on the pipeline length of signals. Ideally, the response to deasserting `from_host_advance` or asserting `dma_to_host_almost_full` would be immediate and would thus require a smaller FIFO depth. Longer pipeline logic requires deeper FIFOs.
- `dma_from_host_advance` needs to be asserted for a DMA transfer to start.
- DMA ToHost transactions should not return more data than requested. Less data than requested can be returned as long as the EOF bit is set on the last cycle of data. Transactions do not need to be subdivided for PCIe payloads, address boundaries, etc. — the DMA engine will handle packet subdivision automatically.
- DMA FromHost transactions may be split into multiple transactions by the DMA engine according to buffer sizes, allowable payloads, transaction throttling, address alignment, etc.
4.4.7 Demand-Mode ToHost DMA

**Figure 19 - Split IF Demand-Mode ToHost DMA Beginning**

- `dma_to_host_valid` is asserted at start of transfer, and remains asserted while transfer data/control is active.
- The 2-qword descriptor is transferred on the first two cycles – bit [4] should be asserted to designate descriptor data.
- If the data width of `dma_from_host_data` is greater than 64, all bits above bit 64 will be invalid/unused during the descriptor cycles.
- Data can begin immediately after 2-qword descriptor.

**Figure 20 - Split IF Demand-Mode ToHost DMA Valid**

- `dma_to_host_valid` should be deasserted when ToHost data is invalid; data does not need to be continuous.

**Figure 21 - Split IF Demand-Mode ToHost DMA Throttling**

- If `dma_to_host_almost_full` is asserted, data should not advance and `dma_to_host_valid` should be deasserted.
Figure 22 - Split IF Demand-Mode Tohost DMA Ending

- dma_from_host_valid is deasserted after last valid data.
- dma_to_host_ctrl[3] should be asserted on last valid data.
- The ToHost acknowledge descriptor will be returned on dma_from_host_data when the DMA engine processes the last dma_to_host_data cycle.
- dma_from_host_ctrl[4] will be asserted for a ToHost acknowledge descriptor
- dma_from_host_ctrl[5] will be asserted on the second cycle of the ToHost acknowledge descriptor

4.4.8 Demand-Mode FromHost DMA

Figure 23 - Split IF Demand-Mode FromHost DMA Beginning

- dma_to_host_ctrl[4] is asserted during 2-qword transfer
- dma_to_host_ctrl[3] is asserted on second cycle of 2-qword transfer
- dma_from_host_ctrl[3] is not asserted, designating this as a FromHost descriptor with data following.
- dma_from_host_ctrl[5] is asserted on second cycle of 2-qword transfer
Figure 24 - Split IF Demand-Mode FromHost DMA Throttling

- `dma_from_host_advance` allows the write to continue; deasserting this signal prevents write data from advancing.
- `dma_from_host_valid` may be asserted for several clock cycles after `dma_from_host_advance` is deasserted due to pipelining delays.

Figure 25 - Split IF Demand-Mode FromHost DMA Ending

- The transaction is considered officially over when the full length of data has been transferred. There is no last-cycle indicator.
- Note that the last cycle of data may have a different dword enable value (pictured above) depending on the length of data to be transferred.

4.4.9 Demand-Mode Gotchas

- All FromHost data and FromHost/ToHost acknowledge descriptors will be returned through the design using the full 64-bit address space. If the high address is not set correctly for a demand-mode design, the descriptors and data may be returned to the wrong place. For example, a demand-mode design in FPGA B would need to set high address descriptors with an address set to 0x01000000. Failing to do so would mean that descriptors and data would be returned to the default address of 0x00000000, which would be directed to FPGA A.
- ToHost transactions need to transfer the exact amount of data specified in the length field of the descriptor – more or less will cause hangs or undefined behavior.
4.5 ‘Combined’ Interface
The ‘Combined’ interface uses simple address/data ports for target, DMA0, DMA1, and DMA2 transactions. Users may want to use this interface if each of the DMA engines access the same memory space, if connecting DMA data to a processor-like interface (AXI), or if Demand-mode DMA is not needed. In addition, this interface is compatible with that on the pcie_target_dma_mux module in legacy V5/V6/V7 designs, and is used by the example userFPGA MainRef design. The Combined interface Verilog module is located in the reference materials at common/pcie/pcie_dma/user_fpga_ultrascale/dini_interface_combined.v. The module attaches to a DiniBus interface from the PCIe netlist or TMB connection; the DiniBus ports will not be described below – see DiniBus section for signal descriptions.

Table 8 – Combined Interface Parameter Definitions

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA_WIDTH_IN</td>
<td>Width of internal DiniBus Interface data. Should always be set to 256.</td>
</tr>
<tr>
<td>DATA_WIDTH_OUT</td>
<td>Width of Split Interface data. Set to 64 for compatibility with legacy designs. Valid values are 64, 128, and 256.</td>
</tr>
<tr>
<td>TAG_WIDTH_OUT</td>
<td>Width of DMA tags. Will be 8 for 64-bit data, 10 for 128-bit data, and 14 for 256-bit data.</td>
</tr>
<tr>
<td>NUM_DMA_ENGINES</td>
<td>Number of DMA engines in the design. Should always be set to 3.</td>
</tr>
<tr>
<td>ONECLOCK</td>
<td>Set to 1 if DiniBus and Combined Interfaces run on the same clock (pcie_clk and user_clk are the same signal). Set to 0 otherwise (default).</td>
</tr>
<tr>
<td>NUM_PORTS</td>
<td>Number of unique ports for the user design. The port selected for a transfer will depend on the decode bits.</td>
</tr>
<tr>
<td>INTERFACE DECODE_LOWER_BIT</td>
<td>Lower bit (inclusive) to use for the port selection address decode.</td>
</tr>
<tr>
<td>INTERFACE DECODE_UPPER_BIT</td>
<td>Upper bit (inclusive) to use for the port selection address decode.</td>
</tr>
<tr>
<td>PORT_DISABLE</td>
<td>One-hot signal that will disable a port and remove the resources associated with it. Useful if address map has holes in it, and not all port address decodes are connected to actual user design. Default to 0 for all ports enabled.</td>
</tr>
<tr>
<td>DMA_ENGINE_ENABLES</td>
<td>One-hot enable signal for DMA engines. Default to 3'b111 for all engines enabled.</td>
</tr>
</tbody>
</table>
**UPPER_ADDRESS_MASK**

Only allow read/write enable to assert for transactions that have an address that satisfies the following equation:

\[(\text{address}[63:32] \& \text{UPPER_ADDRESS_MASK}) == \text{upper_address_check}[31:0]\]

Allows for multiple combined interfaces to use the same dinibus interface. By default, set to 0 and set `upper_address_check` to 0.

**ENABLE_BROADCAST**

When enabled, allow read/write enable to assert for transactions that have an address that satisfies the following equation:

\[(\text{address}[63:32] \& \text{UPPER_ADDRESS_MASK}) == \text{UPPER_ADDRESS_MASK}\]

**MAX_OUTSTANDING_READS**

Number of read requests (read enable pulses) that can be pending. Max value of 512 – must be smaller than internal toHost FIFO depth, and will set the almostfull flag limit for the toHost FIFOs. Recommended value of 128.

**NUM_USER_INTERRUPT**

Number of user interrupts on the interface.

**USER_INTERRUPT_SOURCE_ADDRESS**

64-bit origin address to use for user interrupts.

**TARGET_TIMEOUT_VALUE**

Number of clock cycles to wait for a target read transaction to return before returning to an idle state. Value must be less than the BAR0 timeout register, defaults to 0x7FF (1K clock cycles).

---

**Table 9 – Combined Interface Signal Definitions**

<table>
<thead>
<tr>
<th>Signal</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>user_clk</td>
<td>ToHost</td>
<td>User-supplied clock for the combined interface signals listed below.</td>
</tr>
<tr>
<td>user_reset</td>
<td>ToHost</td>
<td>User-supplied synchronous reset for the combined interface signals listed below.</td>
</tr>
<tr>
<td>interface_ready [NUM_PORTS-1:0]</td>
<td>ToHost</td>
<td>One-hot port ready. When enabled, transactions are allowed to flow to the port. Should be treated as an almostfull signal – transactions may continue to flow for a short period after deassertion.</td>
</tr>
<tr>
<td>write_enable [NUM_PORTS-1:0]</td>
<td>FromHost</td>
<td>One-hot port write enable. When enabled, write address, data, and byte enables are valid.</td>
</tr>
<tr>
<td>write_address [63:0]</td>
<td>FromHost</td>
<td>Address to use for the current write transaction.</td>
</tr>
<tr>
<td>write_data [DATA WIDTH_OUT-1:0]</td>
<td>FromHost</td>
<td>Data to use for the current write transaction.</td>
</tr>
<tr>
<td>write_be [DATA WIDTH_OUT/8-1:0]</td>
<td>FromHost</td>
<td>Byte enables for the current write transaction.</td>
</tr>
<tr>
<td>Field</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>read_enable</td>
<td>FromHost</td>
<td>One-hot port read enable. When enabled, read address, tag, and byte enables are valid.</td>
</tr>
<tr>
<td>read_address</td>
<td>FromHost</td>
<td>Address to use for the current read transaction.</td>
</tr>
<tr>
<td>request_tag</td>
<td>FromHost</td>
<td>Tag to be saved and returned with valid read data.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[0] First cycle of transfer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[1] Last cycle of transfer (EOF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[2] Dword[0] valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[5:4] Transfer type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2'b00: Target</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2'b01: DMA0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2'b10: DMA1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2'b11: DMA2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[7:6] Dword offset</td>
</tr>
<tr>
<td>read_be</td>
<td>FromHost</td>
<td>Byte enables for current read transaction.</td>
</tr>
<tr>
<td>read_data</td>
<td>ToHost</td>
<td>Read return data.</td>
</tr>
<tr>
<td>read_data_tag</td>
<td>ToHost</td>
<td>Tag returned with data.</td>
</tr>
<tr>
<td>read_data_valid</td>
<td>ToHost</td>
<td>Valid signal for return data.</td>
</tr>
<tr>
<td>user_interrupt</td>
<td>ToHost</td>
<td>Active-high interrupt signal from the user design to the PCIe core.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Interrupts can either be pulses or levels when in legacy mode, and must be pulses in MSI mode</td>
</tr>
<tr>
<td>upper_address_check</td>
<td>ToHost</td>
<td>Used in conjunction with UPPER_ADDRESS_MASK to check if the 64-bit address is valid for this module.</td>
</tr>
<tr>
<td>debug_write_data_count</td>
<td>FromHost</td>
<td>Count number of cycles of write data.</td>
</tr>
<tr>
<td>debug_read_data_count</td>
<td>FromHost</td>
<td>Count number of cycles of read data.</td>
</tr>
</tbody>
</table>
4.5.1 FromHost Write Transactions

Figure 26 – Combined IF FromHost Write

- Address, data, and byte enables are always delivered on a single clock cycle, and are valid only when write_enable is active
- One transaction may happen each cycle
- Deasserting interface_ready will prevent write_enable from asserting after several cycles of turnaround – treat interface_ready as an almost-full signal to the module

4.5.2 ToHost Read Transactions

Figure 27 – Combined IF ToHost Read

- Address, byte enables, and tag are always delivered on a single clock cycle, and are valid only when read_enable is active
- One read request may happen each cycle
- Deasserting interface_ready will prevent read_enable from asserting after several cycles of turnaround – treat interface_ready as an almost-full signal to the module
- The request_tag needs to be stored and returned when read_data_valid is asserted.
- There is no latency requirement on the interface, and data may be returned an indeterminate time after being requested. However, if the read request was the result of a BAR transaction, the PCIe transaction will time out after 4K clock cycles. If data is returned after this time it may cause issues on the PCIe link.
5 Interrupts

5.1 DMA Interrupts
All DMA descriptors have an option bit to generate an interrupt upon completion of the descriptor. By default, all descriptors created by the AETest driver will enable interrupts, and the driver will use the generated interrupts as indicators to update the descriptor list pointers and manage buffers. Turning off interrupts will prevent the driver from functioning properly.

5.2 User Interrupts
The user_interrupts port should be used to signal that an event has happened in user-space and trigger the interrupt handler in the driver. The number of inputs to the interface is defined as a parameter in the user interface module. Interrupt inputs can either be pulses (at least one user_clock cycle in length) or can be held in an active state. The port is active high.

One interrupt will be issued to the PCIe core for every set of interrupts that is received from the user in the time between interrupt-handler calls. For this reason, all active interrupts must be handled on each interrupt-handler call; a new interrupt will not be issued for any interrupt that was previously not handled. For further details, please reference the driver code.

There are a maximum of 32 user-interrupt signals viewable in the BAR0 register. Signals from the user modules are combined so that each FPGA’s interrupts show up as a single signal in the BAR0 register. For example, any user_interrupt signal asserted in the user interface module in FPGA A will result in bit [0] being asserted; FPGA B will assert bit [1], and so on.

If the passthrough register is set, the user interrupt will not be latched and the interrupt condition will clear as soon as the user FPGA condition is cleared. This allows the mask and interrupt control logic to reside completely in user space, but spurious interrupts may occur after the interrupt condition is cleared but the assertion has yet to flush from the interrupt pipeline.

5.3 Software
The AETest function wait_on_interrupt() provides the interface for software to wait for an interrupt event to happen. The function will handle both DMA interrupts and user interrupts. This function may return for no reason, or may return multiple times for each interrupt, or may return immediately when called. Multiple interrupts may only cause one function call to return. However, an interrupt set by the hardware will always cause at least one wait_on_interrupt to trigger.

```
uint64_t m_DiniProducts::wait_on_interrupt(uint64_t mask, int timeout_seconds);
```

Description: Block until an interrupt is set by the hardware.
Parameters:
uint64_t mask: {user_interrupt[31:0], 23’h0, dma_interrupt[8:0]} One-hot encoded mask of interrupts to wait on. Lower 32 bits correspond to DMA interrupts, in the same order as the BAR0 register. Upper 32 bits correspond to the user interrupts. Multiple bits can be set, in which case the function will return when any of the interrupts specified in the mask are triggered.

int timeout_seconds: Number of seconds to wait for an interrupt. Function will return automatically at the end of the timeout. If a number less than or equal to 0 is specified, the function will wait infinitely for the interrupt.

Return value: 64-bit mask of the interrupts that were requested that have triggered. If a timeout has occurred, interrupt mask will be 0.

When waiting on a user interrupt, the intended order of events is as follows:

1) User software calls wait_on_interrupt(), which has the driver clear interrupt status and set BAR0 user interrupt mask. User software will block until wait_on_interrupt() returns.
2) RTL design triggers interrupt on user_interrupt port. Either a pulse or a static level is acceptable on the user_interrupt port.
3) The driver will read the BAR0 registers and determine that the user interrupt took place. It will clear the BAR0 mask bit for that interrupt so that no more interrupts will be generated until the next wait_on_interrupt() call for that interrupt. It will pass the triggered interrupt mask to wait_oninterrupt().
4) wait_on_interrupt() will return and the user software will receive the triggered interrupt mask. If the RTL is using static levels, the user software should clear the interrupt in the user design.

Gotchas:
1) Always use the triggered interrupt mask to check if an interrupt has actually occurred.
2) If RTL design has triggered the user interrupt port with a pulse before the wait_on_interrupt() call executes, the interrupt will be missed.

5.4 MSI Message Vectors
The PCIe endpoint will request 32 MSI message vectors. Depending on the number of messages allocated by the system, messages will convey the following information:

0 Vectors:
No message passed
1 Vector:
1’h0: DMA interrupt
1’h1: User interrupt
4 Vectors:
  2'h0: DMA0 interrupt
  2'h1: DMA1 interrupt
  2'h2: DMA2 interrupt
  2'h3: User interrupt

8 Vectors:
  3'h0: DMA0 tohost/fromhost interrupt
  3'h1: DMA1 tohost/fromhost interrupt
  3'h2: DMA2 tohost/fromhost interrupt
  3'h3: DMA0 idle interrupt
  3'h4: DMA1 idle interrupt
  3'h5: DMA2 idle interrupt
  3'h6: Hyperpipe interrupt
  3'h7: User interrupt

16 Vectors:
  4'h0: DMA0 tohost interrupt
  4'h1: DMA0 fromhost interrupt
  4'h2: DMA1 tohost interrupt
  4'h3: DMA1 fromhost interrupt
  4'h4: DMA0 idle
  4'h5: DMA1 idle
  4'h6: DMA2 idle
  4'h7: DMA2 tohost interrupt
  4'h8: DMA2 fromhost interrupt
  4'h9: Hyperpipe interrupt
  4'hA: User interrupt [0]
  4'hB: User interrupt [1]
  4'hC: User interrupt [2]
  4'hD: User interrupt [3]
  4'hE: User interrupt [4]
  4'hF: User interrupt [31:5]
32 Vectors:
5’h0: DMA0 tohost interrupt
5’h1: DMA0 fromhost interrupt
5’h2: DMA1 tohost interrupt
5’h3: DMA1 fromhost interrupt
5’h4: DMA0 idle
5’h5: DMA1 idle
5’h6: DMA2 idle
5’h7: DMA2 tohost interrupt
5’h8: DMA2 fromhost interrupt
5’h9: Hyperpipe interrupt
5’hui: User interrupt [0]
5’hui: User interrupt [1]
5’hui: User interrupt [2]
5’hui: User interrupt [3]
5’hui: User interrupt [4]
5’hui: User interrupt [5]
5’hui: User interrupt [6]
5’hui: User interrupt [7]
5’hui: User interrupt [8]
5’hui: User interrupt [9]
5’hui: User interrupt [10]
5’hui: User interrupt [12]
5’hui: User interrupt [13]
5’hui: User interrupt [14]
5’hui: User interrupt [15]
5’hui: User interrupt [16]
5’hui: User interrupt [17]
5’hui: User interrupt [18]
5’hui: User interrupt [19]
5’hui: User interrupt [20]
5’hui: User interrupt [31:21]
6 Performance

Xilinx has a white paper on PCIe performance that discusses encoding, packet efficiency, and other overheads that serves as a good primer when discussing performance issues: http://www.xilinx.com/support/documentation/white_papers/wp350.pdf.

The following performance sections all use a test design implemented on a DNPCIE_40G_KU_LL. The source code for the test design is built from pieces readily available in the board support package, although the exact top-level design is only distributed upon request.

Figure 28 - Performance Testing Design Block Diagram

The results listed below were obtained on a test system with the following qualities:
Processor: Intel(R) Core(TM) i5-4670 CPU @ 3.40GHz
Motherboard: ASRock Fatal1ty Z97 Killer
Memory: 16GB (8GB x2) DDR3-1600 (PC3-12800) CL11-11-11-28

6.1 Bandwidth

The AETest bandwidth tests (available from the AETest DMA menu) fill the descriptor list with transactions of the full DMA buffer size and then count the number of transactions that can be completed each second. No copies are performed on data in Host PC memory to isolate the performance of the PCIe DMA design. Using these tests, the maximum ToHost bandwidth achieved was ~6080 MB/s, and the maximum FromHost bandwidth was 4840 MB/s.

The most common systems use 128-byte payloads, which means that for our 256-bit DiniBus protocol we should see one cycle of control followed by 4 cycles of data. If packets arrive back-to-back, this means that we should have a maximum bandwidth of 250MHz * 32 bytes/cycle * 4/5 cycles = 6400 MB/s. Descriptor fetches and other message passing account for roughly 5% of traffic, so the DMA data bandwidth to expect is ~6080 MB/s. In performance testing, 6080 MB/s was achieved in the ToHost direction. At the time of writing, transactions in the FromHost direction only achieved a maximum of 4840 MB/s; this issue has been traced to the upstream PCIe controller returning data packets with less than the maximum payload, and not
returning packets back-to-back. Investigations are pending as to how to improve performance in the FromHost direction.

**6.2 Latency**
The PCIe Netlist has the following latencies through the RTL design:
- PCIe hard block to user design, target writes: ~100ns
- PCIe hard block to user design, target read request: ~100ns
- User design to PCIe hard block, target read return: ~90ns
- PCIe hard block to user design, DMA data: ~140ns
- User design to PCIe hard block, DMA data: ~160ns

Target reads are issued at an approximate rate of once per microsecond by most host PCs.

The example user design has a field that will keep track of the number of cycles to perform a demand-mode DMA memory fetch. This shows the minimum round-trip latency from the user design in RTL to the host PC memory. This counter will be displayed if the AETest demand-mode test (available from the AETest DMA menu) is performed. Using the test, the average latency was 690ns, with a minimum of 644ns and a maximum of 972ns. Because the RTL latencies account for ~300ns of this delay, we can approximate the PCIe hard block and host memory read access latency at 690-300=390ns. For a quick back-of-the-envelope calculation, we’ll assume that PCIe hard block and memory write access latency is half of the read latency, or 390ns/2=195ns.

The timing for different operations in host PC memory will depend on the attached system, but with the above number, we can begin to build up some latency approximations for different operations:

- A host-memory DMA FromHost data transfer is comprised of two DMA memory fetches from the host PC, which will take 2*690ns = ~1.4us
- A host-memory DMA ToHost data transfer is comprised of a DMA memory fetch from the host PC and then a memory write to the host PC, which will take 690ns + 160ns + 195ns = ~1.1us.
- A demand-mode DMA FromHost data transfer is comprised of a DMA memory fetch, which will take ~.7us
- A demand-mode DMA ToHost data transfer is comprised of a DMA memory write to the host PC, which will take 160ns + 195ns = ~.4us.
6.3 Optimal Software Strategy

6.3.1 Host-Mode DMA
The DMA engines in the FPGA design will prefetch additional descriptors so that they can begin processing a new descriptor immediately after finishing the current descriptor. This means that the descriptor list in host memory needs to constantly have transfers queued on order to maintain the maximum bandwidth for host-mode DMA transfers.

For FromHost transfers, attaining maximum bandwidth is relatively straightforward – simply call the AETest DMA function as quickly as data is available, and the driver will handle any backpressure from the device. If the RTL design cannot process data at the rate that it is received, the buffers in the driver will eventually all be consumed, and the driver will block until a buffer is free. If the RTL is substantially slower than the software and the driver cannot allocate a buffer for 10 seconds, the driver will fail and the transfer will be cancelled. In order to avoid this, software can call the function `dnpcie_dma_get_active_buffers()` to detect the number of buffers that are being used for a DMA engine. There are 64 1MB buffers instantiated by the driver by default, so using 10-20 buffers for each DMA engine is reasonable. Thus, the pseudocode for a FromHost transfer can be:

```c
while(!feof(data_source)) {
    if(dnpcie_dma_get_active_buffers(dma_num) < 20) {
        fread(buffer, 1, size_one_mb_bytes, data_source);
        dmawrite_boardspace_fromhost(dma_num, board_address, buffer, size_oneMb_dwords);
    }
}
```

For ToHost transfers, attaining maximum bandwidth is a bit trickier. The standard `dmaread_boardspace_tohost()` function will not keep the descriptor list queue constantly busy because it blocks until the tohost transfer is complete, which means that each DMA transfer will incur the full startup latency and bandwidth will suffer accordingly. To work around this, the nonblocking version of the function call should be used. When calling the non-blocking function, a set of handles are passed back to the user that must be preserved and passed back to the `waitfinish()` function. Each handle corresponds to a 1MB buffer being used by the driver, and by passing a handle list to the DMA call, multiple buffers can be used for a single ToHost transfer. However, if the size of the transfer is limited to the size of a single buffer, only the first position in the handle list will contain a valid handle. We can then maintain multiple handle lists to keep track of the number of outstanding ToHost transactions. The pseudocode for a ToHost transfer can be:

```c
void* tohost_handle[NUM_DMA_BUFFERS];
int wr_buffer=0, rd_buffer=0;
int total_transfer_dwords=0, dma_eof=0;
while(!dma_eof && (total_transfer_dwords<requested_transfer_dwords)) {
```
// If no current descriptor in handle list, start a new one
if(tohost_handle[wr_buffer][0]==NULL) {
    dmaread_boardspace_tohost_nonblocking(dma_num, board_address, buffer,
        size_one_mb_dwords, tohost_handle[wr_buffer]);
    wr_buffer = (wr_buffer+1)%20;
}
// Check if finished
int dwords_read = 0;
dmaread_boardspace_tohost_waitfinish(tohost_handle[rd_buffer], &dma_eof, 0
    &dwords_read);

// If data is returned, process it
if(dwords_read>0) {
    /* Process 1MB data chunk here! */
    Tohost_handle[rd_buffer][0]=NULL;
    rd_buffer = (rd_buffer+1)%20;
    total_transfer_dwords += dwords_read;
}

6.3.2 Demand-Mode DMA
The DMA engines will perform whatever memory accesses that the user RTL design initiates
through demand-mode descriptors, so algorithm implementation is largely up to the user.  All
algorithms will start with Software allocating buffers and then transferring the physical
addresses of those buffers to the user RTL design.

For FromHost transfers, it is recommended for Software to copy data-to-be-transferred into
one or more buffers, send the physical addresses of those buffers to the RTL design, and then
wait for the transfer to finish.  Waiting for the transfer to finish can be accomplished with
interrupts, or by spinlocking on a ‘complete’ bit in the buffer space.  For example, your design
could designate that the first byte of a buffer would be written by the user RTL when the
transfer was complete, and software could wait for that event to occur.

For ToHost transfers, it is recommended that Software wait for an interrupt or spinlock on a
specific byte in a buffer.  When the user RTL writes into that buffer, it would write the
‘complete’ byte last and/or signal an interrupt at the end of transfer.  Software can then
process the data buffer and write back to the RTL design to signal that the buffer is available for
another transfer.

When waiting for a transfer to complete, spinlocks will offer lower latency than interrupts, but
come at the cost of processor utilization.
Appendix A: Simulation Testbench

It is highly recommended to simulate the interface between the user RTL design and the PCIe DMA design to ensure correct operation. A PCIe behavioral model is provided with convenient tasks for emulating the behavior of the full PCIe DMA design. This behavioral model is simple and fast because it does not require simulating a host PC PCIe root complex and the transceivers for the physical PCIe interface. An encrypted Verilog simulation netlist is also available for users that wish to simulate the exact PCIe DMA code, but this will require that the user creates a testbench with a host PC PCIe root complex to communicate to the PCIe DMA design (not included).

Files for configFPGA

`common/pcie/pcie_dma/pcie_fpga/testbench_ultrascale/pcie_config_sim_model.v`
Simulation model of the configFPGA design. Tasks are used for stimulating the user interfaces. The testbench is configurable to connect to a user-specified number of interfaces.

`common/pcie/pcie_dma/pcie_fpga/testbench_ultrascale/pcie_board.v`
Board-level testbench connections between configFPGA, appropriate IO modules, and the user design.

`common/pcie/pcie_dma/pcie_fpga/testbench_ultrascale/tb_top.v`
Top-level testbench that instantiates the board-level testbench and calls the Target/DMA tasks.

`common/pcie/pcie_dma/pcie_fpga/testbench_ultrascale/bar_tests.v`
Example list of Target interface tests to run.

`common/pcie/pcie_dma/pcie_fpga/testbench_ultrascale/dma_tests.v`
Example list of DMA interface tests to run.

Files for User FPGA

`common/pcie/pcie_dma/user_fpga_ultrascale/dini_interface_combined_if.v`
Translates between pcie tohost/fromhost interface and recommended Target/DMA interface.

`common/pcie/pcie_dma/user_fpga_ultrascale/dini_interface_split_if.v`
Translates between pcie tohost/fromhost interface and recommended Target/DMA interface.

`common/pcie/pcie_dma/user_fpga_ultrascale/pcie_user_design_split_if.v`
BlockRAM attached to Target and DMA interfaces. User should use transaction logic from this design.
**PCIe Simulation Model**

The tasks in the configFPGA simulation model are used to initiate transactions to the user design.

**TSK_TARGET_WRITE**: 32-256 bit write to the target interface
- **input [63:0] address**: 64-bit address to write using Target interface
- **input [3:0] length**: Length of Target write in dwords
- **input [255:0] data**: Data to write

**TSK_TARGET_READ**: 64-bit read from target interface
- **input [63:0] address**: 64-bit address to read using Target interface
- **input [3:0] length**: Length of Target read in dwords
- **output [255:0] data**: Read data returned

**TSK_DMA_WRITE**: Write data from file to DMA interface
- **input [1:0] engine**: DMA engine to use for transfer. Valid engines are 0,1,2.
- **input [63:0] address**: 64-bit address to write using the DMA interface
- **input [23:0] length**: Length of DMA write in dwords
- **input [*:0] buffer_flat**: Buffer of data to send

**TSK_DMA_READ**: Read data from a DMA interface (blocks until data returns)
- **input [1:0] engine**: DMA engine to use for transfer. Valid engines are 0,1,2.
- **input [63:0] address**: 64-bit address to read using the DMA interface
- **input [23:0] length**: Length of DMA read in dwords
- **output [*:0] buffer_flat**: Buffer of data read

**TSK_DMA_READ_NONBLOCKING**: Request read data from a DMA interface and return immediately
- **input [1:0] engine**: DMA engine to use for transfer. Valid engines are 0,1,2.
- **input [63:0] address**: 64-bit address to read using the DMA interface
- **input [23:0] length**: Length of DMA read in dwords

**TSK_WAIT_DMA_FINISH_ENGINE**: Wait for DMA engine to finish all transactions, and return read data if waiting on a nonblocking read.
- **input [1:0] engine**: DMA engine to use for transfer. Valid engines are 0,1,2.
- **output [*:0] buffer_flat**: Buffer of data read

**TSK_WAIT_DMA_FINISH**: Wait for all DMA transfers to finish

**Notes:**
1) For target accesses, BAR1 will be used to mimic the default AETest APIs
2) Target data will be automatically aligned to the provided address. Return data will be aligned into the LSBs of the register.
3) The combination of target address and length should not cross a 256-bit boundary.

TSK_DEMANDMODE_DMA : Set up the reference design to initiate DMA transfers
input [1:0] engine : DMA engine to use for transfer. Valid engines are 0,1,2.
input [63:0] board_address : Board address for DMA transfer
input [63:0] host_address : Host address for DMA transfer
input [4:0] first_be : First word byte enable
input [4:0] last_be : Last word byte enable
input [23:0] dword_length : Length of DMA transfer in dwords
input direction_tohost : DMA direction; 0x0=fromhost, 0x1=tohost

Note: TSK_DEMANDMODE_DMA requires the use of the split interface and reference design provided in pcie_user_design_ultrascale.v. Users implementing their own logic to initiate DMA transfers will need to modify this function accordingly.

Modeling Software Interaction
The PCIe Simulation Model allows the user to simulate software API calls with the following mapping:

const char* regwrite_boardspace_dword(uint64_t board_address, uint32_t data) :
   TSK_TARGET_WRITE(board_address, 1, data);

const char* regread_boardspace_dword(uint64_t board_address, uint32_t* data) :
   data = TSK_TARGET_READ(board_address, 1);

const char* dmawrite_boardspace_fromhost(uint32_t dma_num, uint64_t board_address,
uint32_t* data, uint32_t size_dwords) :
   TSK_DMA_WRITE(dma_num, board_address, size_dwords, data);

const char* dmaread_boardspace_tohost(uint32_t dma_num, uint64_t board_address,
uint32_t* data, uint32_t size_dwords, uint32_t* dwords_transferred) :
   data = TSK_DMA_READ(dma_num, board_address, size_dwords);
Appendix B: Example User Design
The example user design can be found in the reference materials at FPGA_Reference_Designs/common/pcie/pcie_dma/user_fpga_ultrascale/pcie_user_design_split_if.v. It is intended to attach to the Split interface and fully test all target and DMA features. It serves as a good example for how to attach registers and blockram to the target signals, and how to handle both demand-mode and host-mode DMA transfers.

Figure 29 - User Design Block Diagram

The example user design can run at any of the data widths supported by the Split interface at a frequency up to 250MHz; for this reason, this design is used for performance testing for the DMA engines.

Target transactions will be able to access both BlockRAM and control registers for demand-mode testing. DMA host-mode transactions will access BlockRAM. While the target and DMA-host mode transactions both access BlockRAM, the RAMs are separate and not shared between interfaces. This means that to test that accesses are working correctly, the same interface must be used to write and then read back the RAM. For example, writing to RAM using DMA0 and reading back using DMA0 will give the same data pattern, but writing to RAM using DMA0 and reading back using DMA1, DMA2, or target accesses will not.
### Table 10 - Example User Design Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA_WIDTH</td>
<td>Width of Split Interface data. Set to 64 for compatibility with legacy designs. Valid values are 64, 128, and 256.</td>
</tr>
<tr>
<td>CTRL_WIDTH</td>
<td>Width of DMA tags. Will be 8 for 64-bit data, 10 for 128-bit data, and 14 for 256-bit data.</td>
</tr>
<tr>
<td>MAX_DMA_ENGINES</td>
<td>Number of DMA engines in the design. Should always be set to 3.</td>
</tr>
<tr>
<td>NUM_DMA_ENGINES</td>
<td>Number of DMA engines in the design that will be enabled. DMA engines will be enabled in order, so setting a value of 1 would mean that only DMA0 would be enabled. Max value of 3.</td>
</tr>
<tr>
<td>DEMAND_MODE_RETURN_ADDRESS</td>
<td>64-bit address used to identify where demand-mode completion descriptors should be returned.</td>
</tr>
</tbody>
</table>

### Table 11 - Example User Design Register Address Map

<table>
<thead>
<tr>
<th>Byte Addr</th>
<th>Name</th>
<th>Bit Definitions</th>
<th>Detailed Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 – 0xFFFC RW</td>
<td>BlockRAM</td>
<td>[31:0] BlockRAM</td>
<td>Target-accessible BlockRAM for scratch space testing.</td>
</tr>
<tr>
<td>0x1000 RW</td>
<td>DMA0 Lower Address</td>
<td>[31:0] DMA0 Address [31:0]</td>
<td>Lower 32 bits of the physical address to use for DMA0 demand-mode accesses</td>
</tr>
<tr>
<td>0x1004 RW</td>
<td>DMA0 Upper Address</td>
<td>[31:0] DMA0 Address [63:32]</td>
<td>Upper 32 bits of the physical address to use for DMA0 demand-mode accesses</td>
</tr>
<tr>
<td>0x100C R</td>
<td>DMA0 Debug Status 0</td>
<td>[31:0] Total FromHost count</td>
<td>[31:0] Total number of dwords transferred in the FromHost direction during the test.</td>
</tr>
<tr>
<td>0x1018 RW</td>
<td>DMA0 Total Test Size</td>
<td>[31:0] Size in dwords</td>
<td>Total number of dwords for the demand-mode test.</td>
</tr>
<tr>
<td>Address</td>
<td>Description</td>
<td>Field Details</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0x101C</td>
<td>DMA0 Test Transfer Size</td>
<td>[31:0] Size in dwords</td>
<td>Number of dwords per descriptor for the demand-mode test.</td>
</tr>
<tr>
<td>0x1020 R</td>
<td>Demand-mode ID 0</td>
<td>[31:0] ID</td>
<td>Static identifier for the demand-mode-enabled user design. Register will have a value of 0x00000002.</td>
</tr>
<tr>
<td>0x1024 R</td>
<td>Demand-mode ID 1</td>
<td>[31:0] ID</td>
<td>Static identifier for the demand-mode-enabled user design. Register will have a value of 0x11223399.</td>
</tr>
<tr>
<td>0x1028 R</td>
<td>DMA0 Debug Status 1</td>
<td>[31:0] Total ToHost count</td>
<td>[31:0] Total number of dwords transferred in the ToHost direction during the test.</td>
</tr>
<tr>
<td>0x102C R</td>
<td>DMA0 Debug Status 2</td>
<td>[7:0] FromHost descriptor data count</td>
<td>[7:0] Number of dwords received for the current FromHost descriptor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[15:8] ToHost descriptor data count</td>
<td>[15:8] Number of dwords sent for the current ToHost descriptor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[31:28] Demand-mode test state</td>
<td></td>
</tr>
<tr>
<td>0x1030</td>
<td>DMA1 Lower Address</td>
<td>[31:0] DMA0 Address [31:0]</td>
<td>See DMA0 description</td>
</tr>
<tr>
<td>0x1034</td>
<td>DMA1 Upper Address</td>
<td>[31:0] DMA0 Address [63:32]</td>
<td>See DMA0 description</td>
</tr>
<tr>
<td>Address (Hex)</td>
<td>Register Name</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 0x1038 RW    | DMA1 Control           | [0] Demandmode On  
[1] Read data failed  
[2] Read data started  
[3] Reserved  
[7:4] Last dword byte enables  
[11:8] First dword byte enables | See DMA0 description                                                      |
| 0x103C R     | DMA1 Debug Status 0    | [31:0] Total FromHost count                                                | See DMA0 description                                                  |
| 0x1040 RW    | DMA1 Total Test Size   | [31:0] Size in dwords                                                     | See DMA0 description                                                  |
| 0x104C RW    | DMA1 Test Transfer Size| [31:0] Size in dwords                                                     | See DMA0 description                                                  |
| 0x1048 R     | DMA1 Debug Status 1    | [31:0] Total ToHost count                                                 | See DMA0 description                                                  |
| 0x104C R     | DMA1 Debug Status 2    | [7:0] FromHost descriptor data count  
[15:8] ToHost descriptor data count  
[27:16] Reserved  
[31:28] Demand-mode test state | See DMA0 description                                                  |
| 0x1050 RW    | DMA2 Lower Address     | [31:0] DMA0 Address [31:0]                                                | See DMA0 description                                                  |
| 0x1054 RW    | DMA2 Upper Address     | [31:0] DMA0 Address [63:32]                                               | See DMA0 description                                                  |
| 0x1058 RW    | DMA2 Control           | [0] Demandmode On  
[1] Read data failed  
[2] Read data started  
[3] Reserved  
[7:4] Last dword byte enables  
[11:8] First dword byte enables | See DMA0 description                                                      |
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Field(s)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x105C R</td>
<td>DMA2 Debug Status 0</td>
<td>[31:0] Total FromHost count</td>
<td>See DMA0 description</td>
</tr>
<tr>
<td>0x1060 RW</td>
<td>DMA2 Total Test Size</td>
<td>[31:0] Size in dwords</td>
<td>See DMA0 description</td>
</tr>
<tr>
<td>0x106C RW</td>
<td>DMA2 Test Transfer Size</td>
<td>[31:0] Size in dwords</td>
<td>See DMA0 description</td>
</tr>
<tr>
<td>0x1068 R</td>
<td>DMA2 Debug Status 1</td>
<td>[31:0] Total ToHost count</td>
<td>See DMA0 description</td>
</tr>
<tr>
<td>0x106C R</td>
<td>DMA2 Debug Status 2</td>
<td>[7:0] FromHost descriptor data count</td>
<td>See DMA0 description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[15:8] ToHost descriptor data count</td>
<td>See DMA0 description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[27:16] Reserved</td>
<td>See DMA0 description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[31:28] Demand-mode test state</td>
<td>See DMA0 description</td>
</tr>
<tr>
<td>0x1070 R</td>
<td>DMA0 Latency</td>
<td>[31:0] Latency counter</td>
<td>Counts the number of cycles between sending a demand-mode FromHost descriptor and receiving data from the host PC memory.</td>
</tr>
<tr>
<td>0x1078 R</td>
<td>DMA1 Latency</td>
<td>[31:0] Latency counter</td>
<td>See DMA0 description</td>
</tr>
<tr>
<td>0x1080 R</td>
<td>DMA2 Latency</td>
<td>[31:0] Latency counter</td>
<td>See DMA0 description</td>
</tr>
</tbody>
</table>

Each DMA engine has a demand-mode state machine capable of writing host memory with an incrementing data pattern, and then reading back/verifying said data pattern. The demand-mode state machine will execute the following steps in order:

**DEMAND_MODE_STATE_IDLE:** Initial state. Will exit when the dma_d Demand_mode_on bit in the control register is set.

**DEMAND_MODE_STATE_WRITE_RETURNADDR0, DEMAND_MODE_STATE_WRITE_RETURNADDR1:** Each state will write 32 bits of the 64-bit return address. This will set the return address for the write completion descriptor that will be returned to the design after the host memory write is completed.

**DEMAND_MODE_STATE_READ_RETURNADDR0, DEMAND_MODE_STATE_READ_RETURNADDR1:** Each state will write 32 bits of the 64-bit
return address. This will set the return address for the read completion descriptor and data that will be returned to the design in response to the host memory read.

**DEMAND_MODE_STATE_WRITE_WRITEDESC**,  
**DEMAND_MODE_STATE_WRITE_WRITEADDR**: Send the two-qword demand-mode ToHost descriptor.

**DEMAND_MODE_STATE_WRITE_WRITEDATA**: Send an incrementing data pattern to write into host memory. The amount of data that is sent per descriptor is determined by the transfer size control register.

**DEMAND_MODE_STATE_WRITE_WAIT**: Wait for the write complete descriptor to be returned to the design. If multiple descriptors are needed to fulfill the total test size control register, the state machine will loop writing descriptors and data.

**DEMAND_MODE_STATE_READ_WRITEDESC**,  
**DEMAND_MODE_STATE_READ_WRITEADDR**: Send the two-qword demand-mode FromHost descriptor.

**DEMAND_MODE_STATE_READ_READDESC**: Wait for the read complete descriptor to be returned to the design. Data will follow immediately after the descriptor.

**DEMAND_MODE_STATE_READ_READDATA**: Read an incrementing data pattern returned from host memory and set the ‘Read data started’ bit. If the data does not match the expected pattern, set the ‘Read data failed’ control bit. The amount of data read per descriptor will match the amount of data written per descriptor, which is set by the transfer size control register.

**DEMAND_MODE_STATE_READ_WAIT**: If multiple descriptors are needed to fulfill the total test size control register, the state machine will loop writing descriptors and reading data. Otherwise, the ‘demand-mode on’ control bit will be cleared and the state machine will return to idle.
Appendix C: Indeterminate-Size ToHost DMA

Both demand-mode and host-mode DMA will support a user RTL design that generates a variable amount of size data that needs to be transferred into host PC memory.

Demand-mode DMA is very suited for this purpose because the user RTL knows the exact amount of data that is generated, and can set up transfers of the correct size to host PC memory.

Host-mode DMA is less suited for this purpose because the host PC does not know how much data to request, and when generating large amounts of data for transfer it’s not viable to have the user RTL wait until it’s done generating data. It’s also inefficient to read out a size register in the user RTL before initiating a DMA transfer, as it’s impossible to keep the descriptor list filled and achieve full bandwidth in this scenario. Instead, host PC software should keep the descriptor list full of request descriptors of the maximum size, and user RTL should use the EOF bit to signal the end of data. In this scenario, the tag returned in user RTL would only set the EOF bit at the very end of data transfer instead of echoing the EOF bit in the request_tag signal or setting the dma_to_host_ctrl EOF bit for each buffer’s worth of data. The software loop for this strategy would be as follows:

```c
void* tohost_handle[20][NUM_DMA_BUFFERS];
int wr_buffer=0, rd_buffer=0;
int dma_eof=0;
while(!dma_eof) {
    // If no current descriptor in handle list, start a new one
    if(tohost_handle[wr_buffer][0]==NULL) {
        dmaread_boardspace_tohost_nonblocking(dma_num, board_address, buffer,
                                             size_one_mb_dwords, tohost_handle[wr_buffer]);
        wr_buffer = (wr_buffer+1)%20;
    }
    // Check if finished
    int dwords_read = 0;
    dmaread_boardspace_tohost_waitfinish(tohost_handle[rd_buffer], &dma_eof, 0
                                          &dwords_read);

    // If data is returned, process it
    if(dwords_read>0) {
        /* Process data chunk here! */
        Tohost_handle[rd_buffer][0]=NULL;
        rd_buffer = (rd_buffer+1)%20;
    }
}
```
Appendix D: Card-to-Card DMA

It is possible to transfer data directly between two cards in the same PCIe system without intervention from software running on the host PC. One card will initiate demand-mode accesses to the physical address of the BAR mappings of the other card. Accesses will then show up as target BAR transactions on the receiving card.

It is recommended that software is used to do the following initialization:
   1) Map BAR addresses of PCIe cards (done in driver)
   2) Transfer physical BAR addresses to user RTL designs
   3) Set up appropriate BAR0 registers for BAR1-4 windowing

Demand-mode descriptors can then be used to transfer data directly to BARs 1-4. BAR0 can also be accessed by demand-mode descriptors to change the BAR1-4 windows, but it’s suggested to do this in software so that any other software-initiated BAR accesses to the card will update the BAR windows correctly.
Appendix E: Troubleshooting/FAQ

Device fails to enumerate
- Check that the FPGA is configured.
- Reboot the machine. If the FPGA was not configured at the time that the Host PC enumerated the devices, then doing a warm reboot of the machine may allow the device to enumerate.
- Check the timing report. Consider any timing violations as fatal, and correct the violations. Most commonly, this requires adding placement constraints; the internals of the design have been verified to synthesize/place/route at full speed. Contact support@dinigroup.com with any questions about failing paths.
- If using a cable interface, check that all sideband signals are driven appropriately. See the schematic for more details. Verify that there are no red LEDs on the cable card, which usually happens if the PRESENT signal is not driven correctly by the FPGA.

Timeouts on Read Data
- Run a simulation to ensure that the user design is responding appropriately to target read requests.
- The default timeout for target reads is set to 4K clock cycles (measured on the internal 250MHz domain). If the user design takes a long time to respond to reads or is on a much slower clock domain, use the BAR0 register to increase this value.

DMA Hang
- Run a simulation to ensure that the user design is responding appropriately to DMA transactions.
- Check OS logs for information from driver.
- Use DMA info option on AETest DMA menu to see current descriptor offsets, descriptor lists, and buffer data.
- If hung on a FromHost transaction, ensure that the user design is not stuck in a state where it is not accepting DMA data.
- If hung on a ToHost transaction, ensure that the user design has provided amount of data requested to the DMA engine.

Driver Fails to Compile/Install
- Contact support@dinigroup.com with a description of the error message and the operating system type/version.
Contact Info

If you have a new feature request or feel that an essential function is missing or broken, please contact support@dinigroup.com.