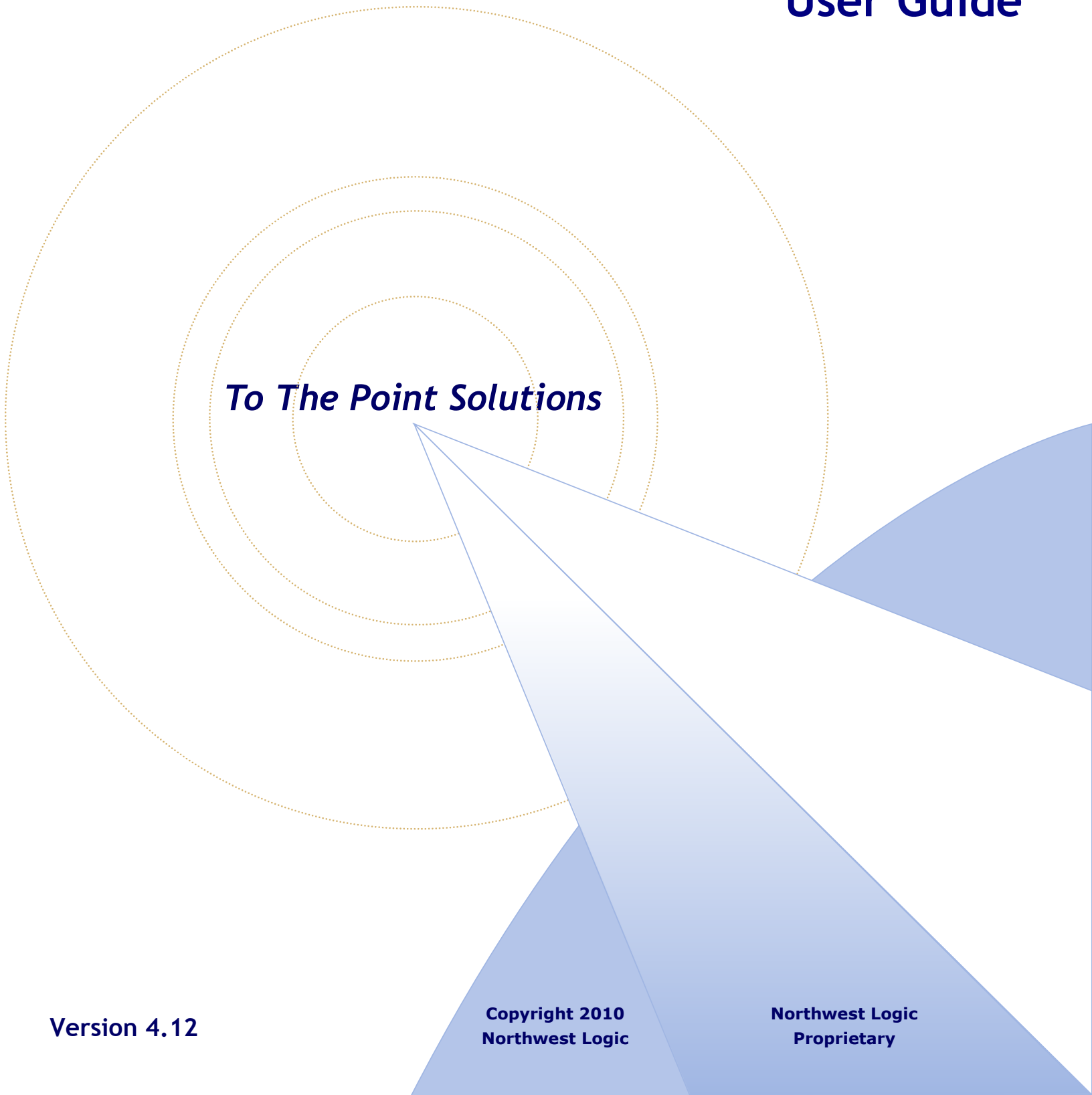
NORTHWEST LOGIC

DMA Back-End Core

User Guide

To The Point Solutions



Version 4.12

1 REVISION HISTORY 4
2 DMA BACK END OVERVIEW 6
2.1 EXPRESSO SOLUTION (PCI EXPRESS 2.0/1.1) 6
2.2 DMA BACK-END CORE FEATURES 8
3 DMA OVERVIEW 10
4 DMA BACK-END CORE ARCHITECTURE 11
4.1 BLOCK DIAGRAM 11
4.2 PORT DIAGRAM 13
5 PCI EXPRESS INTERFACE 15
6 TARGET INTERFACE 16
6.1 TARGET INTERFACE PORT DESCRIPTIONS 16
6.2 TARGET INTERFACE EXAMPLE TRANSACTIONS 23
7 REGISTER INTERFACE 25
7.1 REGISTER MAP 25
7.2 REGISTER INTERFACE PORT DESCRIPTIONS 26
7.3 REGISTER INTERFACE EXAMPLE TRANSACTIONS 28
8 DMA INTERFACE 29
8.1 DMA ENGINE OPERATION 29
8.2 DMA DESCRIPTOR DEFINITION 31
8.2.1 Packet DMA Descriptor Format 32
8.2.2 Block DMA Descriptor Format 41
8.3 DMA INTERFACE PORT DESCRIPTIONS 43
8.4 DMA INTERFACE EXAMPLE TRANSACTIONS 55
8.5 COMMON DMA REGISTER BLOCK 59
8.6 DESCRIPTOR ENGINE 61
8.7 DESCRIPTOR UPDATE ENGINE 61
8.8 DMA REGISTERS 62
8.8.1 Packet DMA Registers 62
8.8.2 Block DMA Registers 71
8.9 ABORTING AND RESETTING DMA 76
8.9.1 Packet DMA 76
8.9.2 Block DMA 77
9 PCI EXPRESS ERROR HANDLING 78
9.1 ERROR HANDLING 78

9.2 ERROR REPORTING INTERFACE 79
10 DMA DIRECT CONTROL INTERFACE 80
10.1 DMA DIRECT CONTROL INTERFACE PORT DESCRIPTIONS 80
10.2 DMA DIRECT CONTROL INTERFACE EXAMPLE TRANSACTIONS 83
11 MANAGEMENT INTERFACE 84
11.1 MANAGEMENT INTERFACE PORT DESCRIPTIONS 84
12 MASTER INTERFACE 87
12.1 MASTER INTERFACE PORT DESCRIPTIONS 87
12.2 MASTER INTERFACE EXAMPLE TRANSACTIONS 89
13 CONFIGURATION SPACE 90
14 MIGRATING FROM PREVIOUS DMA BACK-END VERSIONS 90
15 PACKET DMA REFERENCE DESIGN 91
15.1 PACKET STREAMING INTERFACE PORT DESCRIPTIONS 91
15.2 CARD TO SYSTEM PACKET STREAMING INTERFACE TIMING DIAGRAMS 93
15.3 SYSTEM TO CARD PACKET STREAMING INTERFACE PORTS 94
15.4 SYSTEM TO CARD PACKET STREAMING INTERFACE TIMING DIAGRAMS 96
15.5 CARD TO SYSTEM PACKET GENERATOR 97
15.6 SYSTEM TO CARD PACKET CHECKER 100
15.7 PACKET RATE CONTROL 103
16 ADDRESSED CARD MEMORY DMA REFERENCE DESIGN 104
17 RAM USAGE 105
18 SPEED & SIZE 106
19 FREE EVALUATION CORE 106
20 FOR MORE INFORMATION 106

1 Revision History

This section tracks revisions made to this document by version number

Revision	Date	Changes
3.10	02/09/2009	• Added revision history section to the document.
3.11	06/01/2009	• Added notes to Sections 2.1 and 2.3 indicating which soft and hard PCIe Cores are supported • Updated MSI_En location for NW Logic PCIe 2.0 Core.
4.00	08/18/2009	• Very major update with lots of new functionality and differences in operation including port incompatibilities with prior DMA Back-End versions. Recommend a thorough review. • Added new DMA configuration option, Packet DMA, which is optimized for transferring lots of small packets • Assigned name “Block DMA” to the prior existing DMA mode of operation to be able to differentiate between “Packet DMA” and “Block DMA” operation • Merged Packet DMA info previously only contained in DMA Back-End Packet DMA User Guide 0.12; DMA Back-End Packet DMA User Guide is now discontinued since its information is now included in this main DMA Back-End User Guide. • Removed PCI Express/PCI-X/PCI core-specific information from this User Guide and documented the Management Interface which is used to connect to all cores instead.
4.01	09/10/2009	• Defined bit in Common Registers : DMA_Common_Control_and_Status[6] for software to determine when MSI-X interrupt mode is being used. Only relevant for DMA Back-End designs which are customized to support MSI-X. • Added c2s[C-1:0]_cfg_constants: Bit[3] – Enable C2S Overlap
4.02	11/23/2009	• Block DMA Registers: Added DMA status registers to provide additional information on Descriptor and DMA data read completion errors. • Updated Description of s2c[S-1:0]_data_error & register bits associated with DMA errors
4.03	11/24/2009	• Packet DMA Registers: Added DMA status registers to provide additional information on Descriptor and DMA data read completion errors. • Added additional Packet DMA interrupt mode – interrupt on EOP. See Packet DMA Registers.
4.04	01/26/2010	• Added DMA cfg_constants[47] to selectively disable system memory Descriptor updates (generally enabled when Local Descriptor interface is used)
4.05	03/31/2010	• Updated c2s/s2c_cfg_constants[3] to reflect that overlap can be applied both for S2C and C2S Block DMA (prior indicated this bit could be set only C2S Block DMA) • Added note to DMA Direct Control interface to advise users of this interface that c2s/s2c_cfg_constants[47] == Disable_Descriptor_Updates should generally be set to 1.
4.06	04/06/2010	• Added description to DMA Registers to indicate where Completion Timeout errors are recorded when they occur on Descriptor read completions or DMA data read completions. • Minor clarification to Block DMA DMA engine reset procedure
4.07	04/12/2010	• Added Error Handling : Section9 • Removed references to PCI-X and Conventional PCI

Revision	Date	Changes
4.08	05/07/2010	• Clarified c2s_data_first_chain, c2s_data_last_chain description to make it clear that these are unused and output zero for C2S Packet DMA operation
4.09	08/03/2010	• Added information for new Addressable Packet DMA API • Added read byte enables to target interface • Added addressed packet mode to streaming packet interface • DMA Wait Time and DMA Completed Byte Count registers had incorrect addresses listed; fixed
4.10	08/20/2010	• Added note indicating that Packet DMA should be used for all new designs.
4.11	09/07/2010	• Minor clarifying edits
4.12	09/23/2010	• Added RAM Usage section

2 DMA Back End Overview

The DMA Back-End adds high-performance DMA and target functionality to PCI Express front-end cores. The DMA Back-End user ports and operation is the same between different PCI Express Cores allowing users to easily migrate between cores. The DMA Back-End supports both Northwest Logic ASIC/FPGA PCI Express cores and Xilinx and Altera FPGA hardened PCI Express cores.

The DMA Back-End is a key component of Northwest Logic's full, high performance, hardware/software DMA system solution for PCI Express (Expresso Solution).

2.1 Expresso Solution (PCI Express 2.0/1.1)

Northwest Logic's Expresso Solution as shown in Figure 2-1 and Figure 2-2 is designed to accelerate high-performance user designs into production. The Expresso Solution includes the following components:

- PCI Express Core
  - Low-level PCI Express implementation
  - DMA Back-End is delivered with the ports to connect seamlessly to the following PCIe Cores:
    - Northwest Logic Expresso 3.0 Core (ASIC/FPGA soft core; PCI Express 3.0)
    - Northwest Logic Expresso 2.0 Core (ASIC/FPGA soft core; PCI Express 2.0)
    - Northwest Logic Expresso Core (ASIC/FPGA soft core; PCI Express 1.1)
    - Xilinx Virtex-6LXT/SXT PCIe Core (FPGA hard core; PCI Express 2.0)
    - Xilinx Virtex-5LXT/SXT/FXT PCIe Core (FPGA hard core; PCI Express 1.1)
    - Xilinx Spartan-6LXT PCIe Core (FPGA hard core; PCI Express 1.1)
    - Altera Stratix IV GX PCIe Core (FPGA hard core; PCI Express 2.0)
    - Altera Arria II GX PCIe Core (FPGA hard core; PCI Express 1.1)
- DMA Back-End Core
  - High-performance multi-engine DMA
    - Packet DMA
      - Optimized for small and large transfers
      - Built-in support for packetized user data; also supports non-packet applications
      - Enhanced functionality over Block DMA; recommended for new designs
    - Block DMA
      - Optimized for large transfers
      - Compatible with previous DMA Back-End "Gen3" versions
      - Packet DMA is recommended for new designs
  - Simplified Register and Target accesses
- Packet Application Reference Design
  - Packet Streaming FIFOs
  - Packet Generator
  - Packet Checker
- Addressed Card Memory Application Reference Design
  - DMA Interface
    - Example FIFO modules for connecting DMA interfaces to SDRAM controller
  - Multi-Port Front-End Core
    - Enables additional user ports to be easily integrated
  - Memory Controller Core
    - High-performance DDR3/2, DDR, SDR, Mobile DDR/SDR, or RLDRAM II Controller
- PCI Express Verification Suite
  - Enables simple, powerful system simulations
- DMA Driver
  - Works with DMA Back-End Core to implement high throughput DMA operations
- PCI Express GUI
  - Throughput characterization and PCI Express Solution demonstration

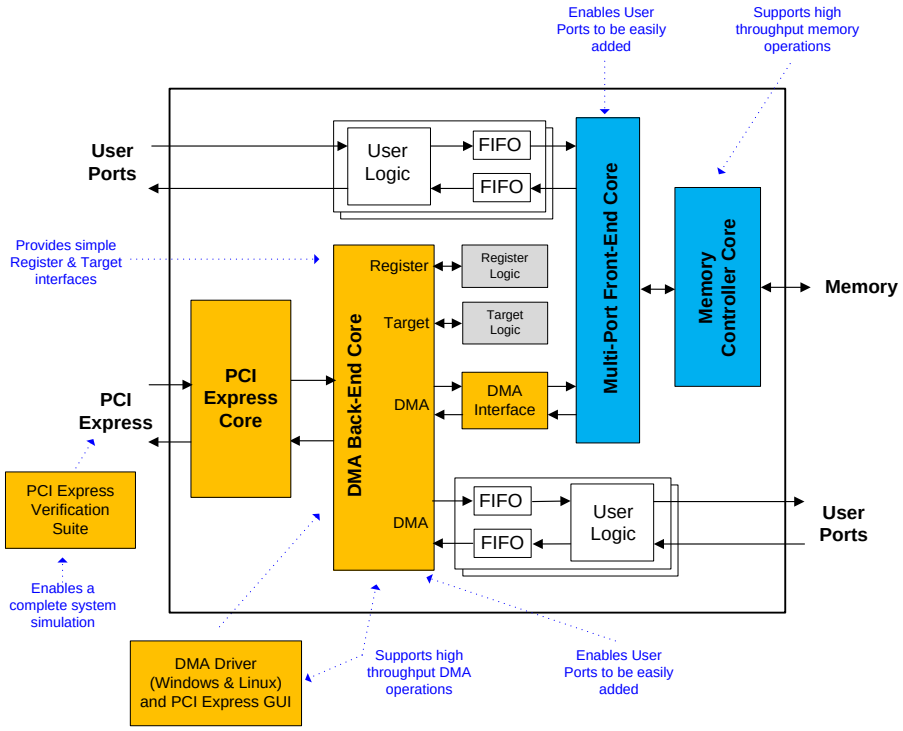


Figure 2-1 Expresso Solution – Addressed Card Memory Application (PCI Express)

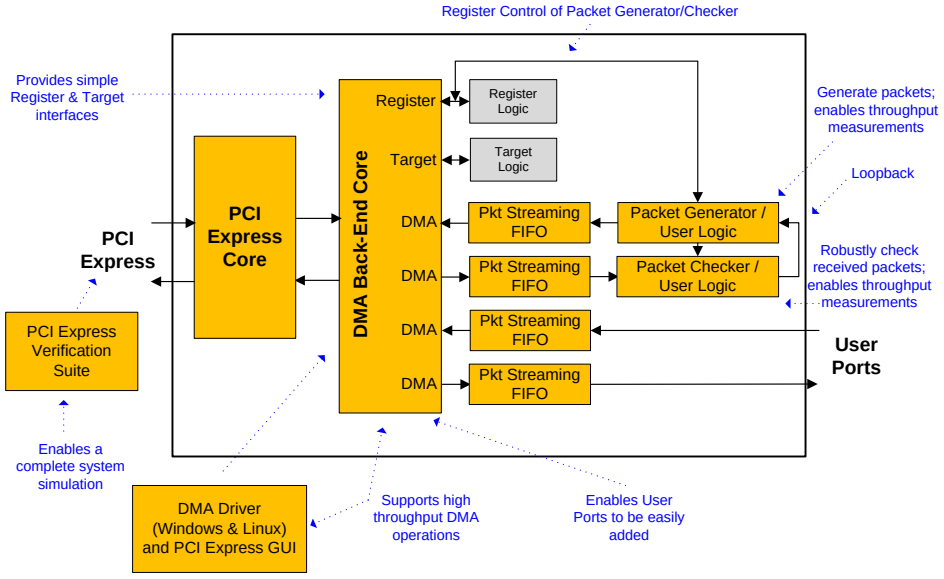


Figure 2-2 Expresso Solution – Packet Application (PCI Express)

2.2 DMA Back-End Core Features

Key features:

- PCI Express Core Support
  - Northwest Logic Expresso 3.0 Core (ASIC/FPGA soft core; PCI Express 3.0)
  - Northwest Logic Expresso 2.0 Core (ASIC/FPGA soft core; PCI Express 2.0)
  - Northwest Logic Expresso Core (ASIC/FPGA soft core; PCI Express 1.1)
  - Xilinx Virtex-6LXT/SXT PCIe Core (FPGA hard core; PCI Express 2.0)
  - Xilinx Virtex-5LXT/SXT/FXT PCIe Core (FPGA hard core; PCI Express 1.1)
  - Xilinx Spartan-6LXT PCIe Core (FPGA hard core; PCI Express 1.1)
  - Altera Stratix IV GX PCIe Core (FPGA hard core; PCI Express 2.0)
  - Altera Arria II GX PCIe Core (FPGA hard core; PCI Express 1.1)
- Three core data width options support a wide variety of applications and device targets
  - 128-bit
  - 64-bit
  - 32-bit
- DMA Interface
  - Very flexible, easy-to-use, high-performance DMA implementation
  - Card-to-System (C2S) DMA Engine
    - Takes data from user logic and makes DMA Write Requests to system memory
    - Demand-driven user interface; user can wait state for flow control
    - Flexible Control - DMA Descriptor Engine fetches DMA Descriptors from a linked list of Descriptors stored in system memory or user logic can directly control the DMA Engine
  - System-to-Card (S2C) DMA Engine
    - Makes DMA Read Requests from system memory, handles the resulting Read Completions, and forwards read data to user logic
    - Demand-driven user interface; user can wait state for flow control
    - Guarantees read data ordering (re-orders completions that were received out of order)
    - Flexible Control - DMA Descriptor Engine fetches DMA Descriptors from a linked list of Descriptors stored in system memory or user logic can directly control the DMA Engine
  - High-performance multi-engine DMA with two software interface options
    - Packet DMA
      - Optimized for small and large transfers
      - Built-in support for packet applications
      - Built-in support for addressed card memory & FIFO applications
      - Enhanced functionality over Block DMA; recommended for new designs
    - Block DMA
      - Optimized for large transfers
      - Built-in support for addressed card memory & FIFO applications
      - Software compatible with previous DMA Back-End “Gen3” versions
      - Mostly user-interface compatible with previous DMA Back-End “Gen3” versions; see Section 14 for migration details.
  - Base Configuration has 1 Card to System DMA Engine and 1 System to Card DMA Engine
  - Multi-Engine Configuration has 1-4 Card to System DMA Engines and 1-4 System to Card DMA Engines
  - DMA Engines are fully independent and are interleaved on a packet basis
  - 32-bit and 64-bit System Address Support
  - 32-bit Descriptor Linked-List System Address Support
  - 64-bit Card Address Support (Block DMA); 36-bit (Packet DMA)
  - Option for user hardware design to provide Descriptors directly
  - MSI and Legacy Interrupt support
  - System address, Card Address, and transfer Byte Count fully support byte alignments allowing for maximum software flexibility
  - Scatter-gather (chaining) DMA engines achieve high throughput even with heavily fragmented system memory
  - Supports extremely long DMA transfers
- Target Interface
  - Very flexible, easy-to-use, high performance, independent target write and read interfaces
  - Supports 32-bit and 64-bit Memory and I/O Base Address Registers

    - Simplifies adding target resources to the user's design
- Register Interface
    - Implements 64KByte 32-bit Memory Base Address Register 0 (BAR0) for DMA and user registers
        - Lower 32 KByte reserved for DMA Back-End registers
        - Upper 32 KByte reserved for user registers
    - Simple, fixed timing Register Interface makes adding user registers trivial
- Master Interface (PCI Express Only)
    - Simple interface supports generation of Memory (32/64-bit address), I/O (Root Port only), Configuration (Root Port only), and Message (Msg/MsgD) transactions
    - Supports write and read transactions with up to one DWORD (32-bit) of payload data
    - Only included for Root Port DMA Back-End applications or on request
- Complete Reference Design Available
    - Illustrates operation of the DMA, Target, and Register interfaces
    - Enables hardware DMA throughput measurements
    - References design is tailed to DMA type:
        - Packet DMA - Packet Applications (see Expresso Solution - Figure 2-2)
            - Packet Streaming FIFO modules to convert Card to System and System to Card DMA Back-End user interfaces into simple packet streaming interfaces; it is recommended to use the provided streaming interfaces for packet applications when possible to simplify the user's design. Please see Section 15.1 for details.
            - Packet Generator generates a wide variety of packet types
                - Generated packet characteristics controlled via BAR0 registers
                - Supports varying packet length, data patterns, and user_status values
                - Supports packet loopback from and then to system memory when paired with a Packet Checker
            - Packet Checker robustly checks a wide variety of packet types
                - Expected packet characteristics controlled via BAR0 registers
                - Supports varying packet length, data patterns, and user_control values
                - Supports packet loopback from and then to system memory when paired with a Packet Generator
            - Example Target Interface logic implements two 32-bit Base Address Registers mapped to the same internal SRAM
            - Example Register Interface logic implements Packet Generator and Packet Checker control registers and PCI Express Receive Buffer credit information (PCI Express only) when available from the underlying PCI Express Core
        - Block DMA - Addressed Card Memory Applications (see Expresso Solution - Figure 2-1)
            - DMA Interface includes FIFOs to convert Card to System and System to Card DMA Back-End user interfaces into simple FIFO interfaces to transfer data to/from SDRAM; it is recommended to use the provided FIFO interfaces for addressed card memory applications when possible to simplify the user's design. Please see Section 16 for details.
            - If not licensing a supported configuration of the Northwest Logic Memory Controller and Multi-Port Front End Cores, the reference design includes a multi-ported internal SRAM in place of the Multi-PortFront-End and Memory Controller cores. The multi-ported SRAM has the DMA Back-End side ports as the cores it replaces and allows for full end-to-end hardware and simulation validation
            - Example Target Interface logic implements two 32-bit Base Address Registers mapped to the same internal SRAM
            - Example Register Interface logic implements PCI Express Receive Buffer credit information (PCI Express only) when available from the underlying PCI Express Core
    - PCI Express Verification Suite
    - Windows 7/Vista/XP Driver and Command Line Test Application
    - Windows 7/Vista/XP GUI Test Application
    - Linux Driver and Test Application
- Source code available
- Customization and Integration services available
- *PCI Express™ Base Specification Revision 2.1/1.1* compliant (backward compatible to 2.0/1.0a)

3 DMA Overview

Direct Memory Access (DMA) is the process of copying large amounts of data efficiently between two devices (typically the host system memory and a bus device) with minimal host processor involvement.

DMA requires a dedicated hardware resource – a DMA Engine – to do the memory copy. The DMA Engine's job is to do the copy operation specified by software. When using a DMA Engine, the software only needs to implement the control function of the copy (tell the DMA Engine where to copy from and to, etc.) rather than having to actually move the data itself to perform the copy.

There are two primary advantages of using DMA

- A DMA Engine is much better at copying memory than software; DMA Engines can issue large burst transactions (as large as the underlying hardware protocol allows) while the processor typically is only capable of very small burst transactions; the protocol efficiency, which is the ratio of payload_transferred / (payload_transferred + hardware_protocol_overhead), is typically extremely poor with the small payload size used by the processor and very good with the larger payload size used by a DMA Engine; a DMA Engine may produce 10 to 100 times greater throughput than a non-DMA software copy
- Software offloads the time-consuming copy task to the DMA Engine and thus frees the processor for other tasks for which software is better suited. The copy operation is a repetitive task requiring only simple decisions and is well suited for hardware acceleration via DMA. Processor resources are better utilized on tasks which software is better suited for such as running the user's applications and processing (converting, parsing, displaying, etc.) the DMA data.

There are many different ways to implement DMA and unfortunately there is no common DMA standard.

4 DMA Back-End Core Architecture

4.1 Block Diagram

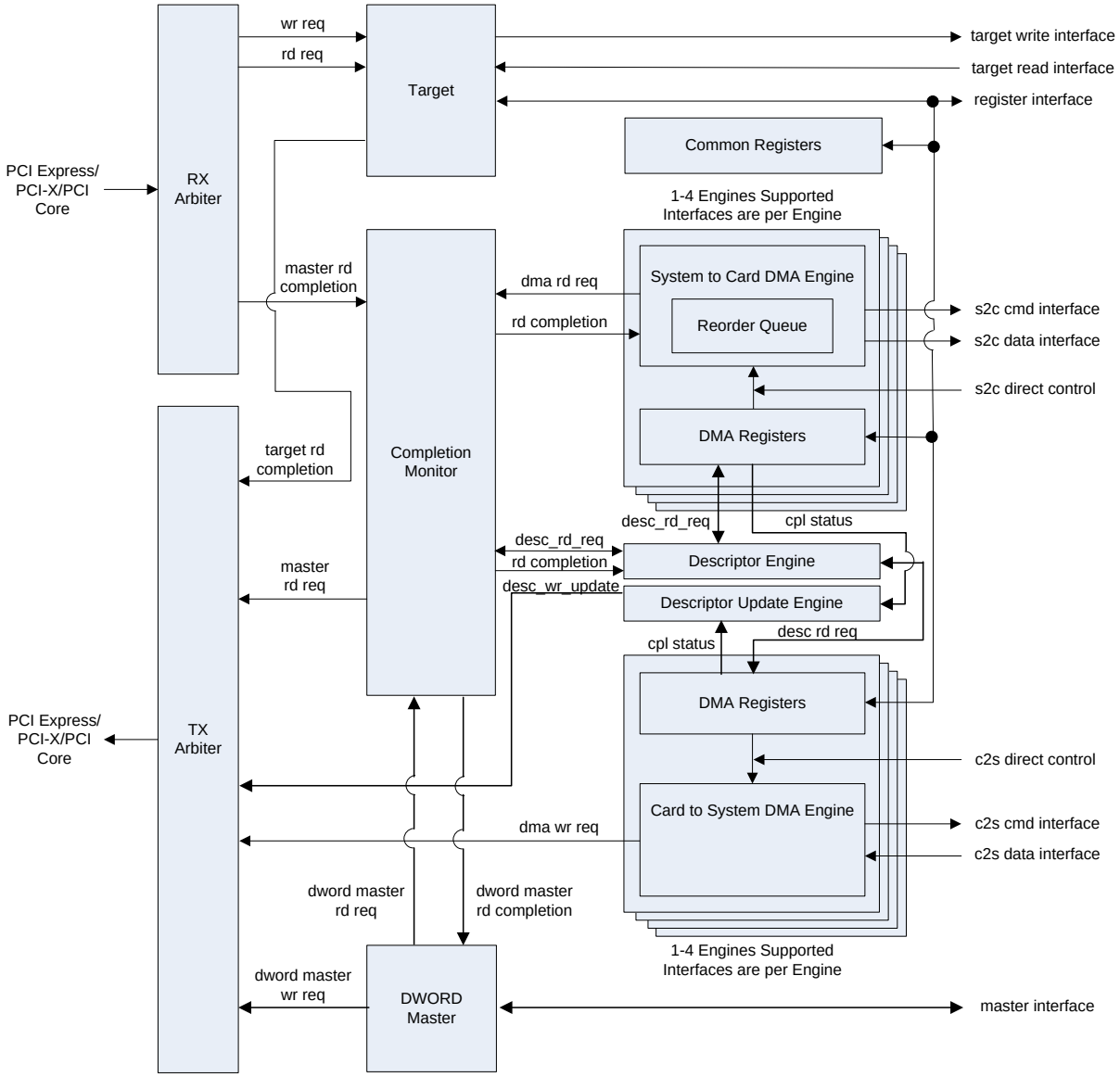


Figure 4-1 DMA Back-End Core Block Diagram

DMA Back-End Core Module Descriptions (module source name in parenthesis)

- RX Arbiter (rx_monitor)
  - Arbitrates inbound traffic
  - Received write requests are forwarded to the Target module for termination on the Target Write Interface or Register Interface
  - Received read requests are forwarded to the Target module for termination on the Target Read Interface or Register Interface
  - Received completions (read data resulting from master read requests) are forwarded to the Completion Monitor module for termination at the appropriate DMA/DWORD Master requestor
- TX Arbiter (be_stream_arb)
  - Arbitrates outbound traffic
  - Transmitted write requests originate from Card to System DMA Engines/DWORD Master
  - Transmitted read requests originate from the Completion Monitor
  - Transmitted completions (read data/status from target reads) originate from the Target module
- Target (target)
  - Generates the Target Write Interface for consuming received write requests
  - Generates the Target Read Interface for satisfying received read request
  - Generates the Register Interface for satisfying received write and read requests targeting the Base Address Register assigned to the DMA Back-End Core and user registers
  - Supports 32/64 Memory Base Address Registers
- Completion Monitor (cpl_monitor)
  - Arbitrates among read requestors to manage limited tag and received completion resources
  - Re-associates received completions with the original request
  - Routes completion data to the original requestor
- DWORD Master (dword_master; PCI Express Only)
  - Completes read, write, and message requests initiated on the Master Interface
  - Processes read completions from read and I/O/Cfg write requests and returns data and status
- Common Registers (common_registers)
  - Implements centralized registers for DMA interrupts and DMA Back-End Core capabilities
- DMA Registers (Packet DMA: dma_registers_packet; Block DMA: dma_registers_block)
  - Implements the DMA registers for one DMA Engine for software to control and to obtain status from the DMA Engine
  - Processes DMA Packets/Chains; makes Descriptor read requests to the Descriptor Engine
  - Makes Descriptor update requests, to Descriptor Update Engine, with Descriptor completion Status (Packet DMA only)
- Descriptor Engine (desc_engine)
  - Centralized resource for fetching Descriptors from system memory
- Descriptor Update Engine (desc_update_engine)
  - Centralized resource for updating Descriptors with DMA completion status (Packet DMA Only)
- Card to System DMA Engine (c2s_dma_engine_pkt)
  - Takes DMA Data from user logic and transmits data to PCI Express using write requests
  - Executes the PCI Express and user logic transactions to fulfill a single Descriptor, returns status, and repeats as long as Descriptors are made available to execute
  - Controlled by DMA Registers or DMA Direct Control Interface
  - Supports 32/64-bit System and Card addresses of any byte alignment
  - Supports byte counts from 1 to 2^32-1 bytes (Block DMA) and 1 to 2^20-1 bytes (Packet DMA) per Descriptor
- System to Card DMA Engine (s2c_dma_engine_pkt)
  - Transmits PCI Express read requests and forwards the resulting read completion data to user logic
  - Executes the PCI Express and user logic transactions to fulfill a single Descriptor, returns status, and repeats as long as Descriptors are made available to execute
  - Controlled by DMA Registers or DMA Direct Control Interface
  - Supports 32/64-bit System and Card address of any byte alignment
  - Supports byte counts from 1 to 2^32-1 bytes (Block DMA) and 1 to 2^20-1 bytes (Packet DMA) per Descriptor
  - Implements a Reorder Queue to ensure that DMA read requests are returned in order (PCI Express Devices are permitted to reorder read transactions)

4.2 Port Diagram

Figure 4-2 and Figure 4-3 groups the DMA Back-End Core ports by function.

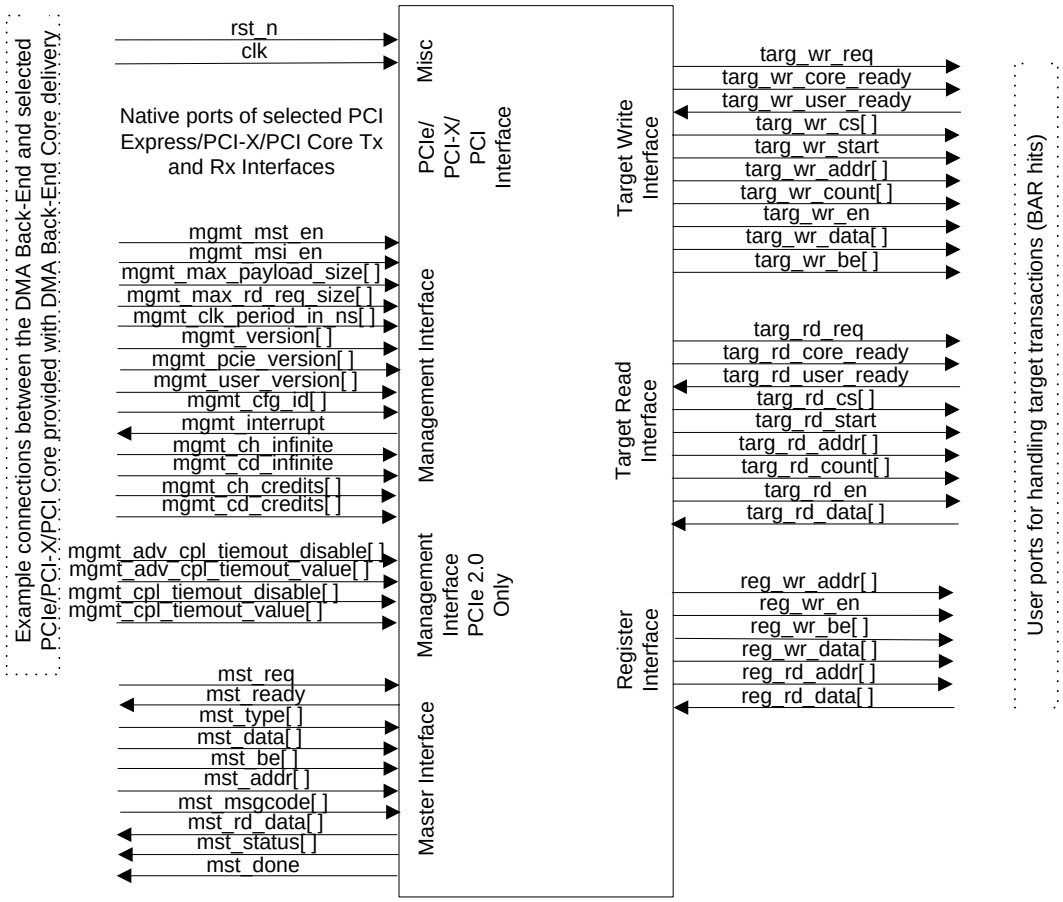


Figure 4-2 DMA Back-End Core Port Diagram (non-DMA ports)

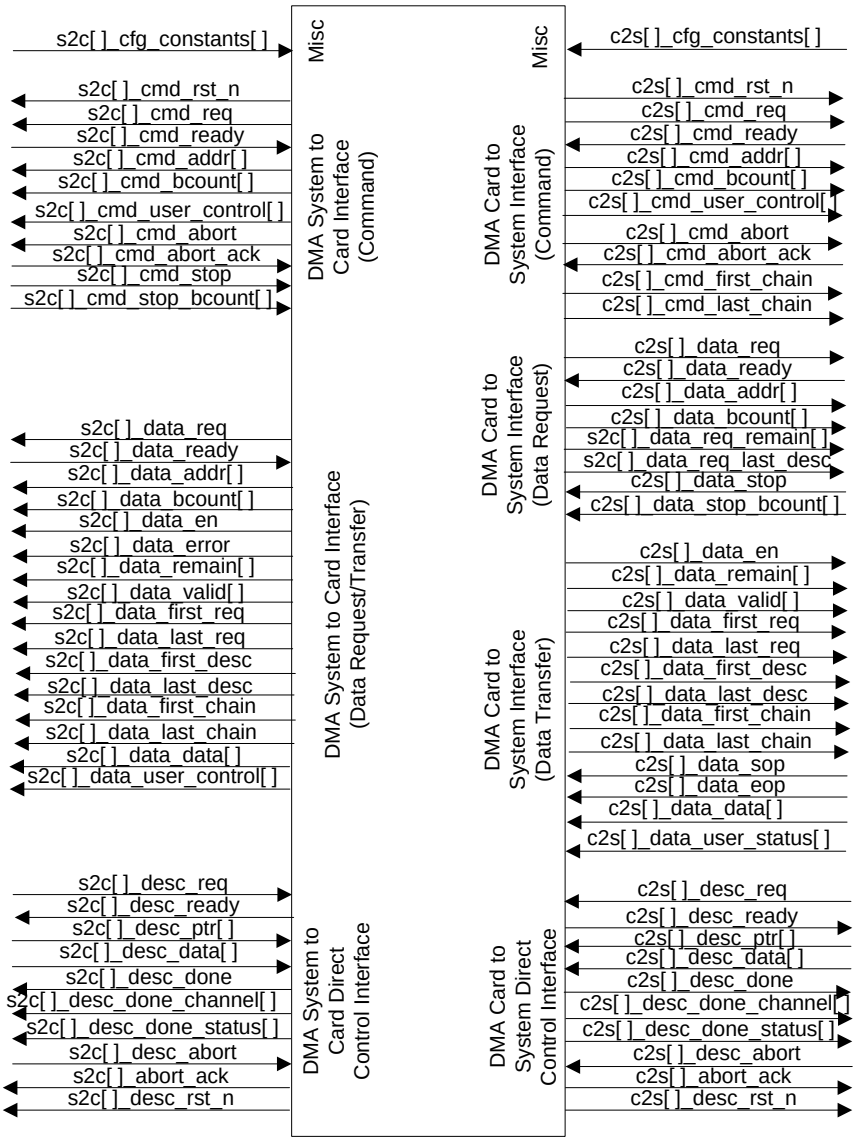


Figure 4-3 DMA Back-End Core Port Diagram (DMA ports)

5 PCI Express Interface

The PCI Express Core implements the required lower-level PCI Express functionality including the Physical Layer, Data Link Layer, and portions of the Transaction Layer. The DMA Back-End Core receives PCI Express Transaction Layer packets from the PCI Express core and transmits PCI Express Transaction Layer packets to the PCI Express core.

Please see the relevant PCI Express core User Guide for information on the PCI Express Interface.

The DMA Back-End is available for numerous PCI Express cores including Northwest Logic and third party hard cores. Please see Section 2.2 for detailed core support.

6 Target Interface

6.1 Target Interface Port Descriptions

The Target Interface consists of two separate interfaces: Target Write Interface and Target Read Interface. Target writes have inbound data and target reads have outbound data. To take advantage of the full-duplex nature of PCI Express, the Target Write Interface and Target Read Interface are independent from one another. It is thus possible for both the Target Write Interface and Target Read Interface to be active at the same time.

Received write requests (and associated write data) that map to BAR1-5 are completed using the Target Write Interface (targ_wr_* ports). For received target write requests, the DMA Back-End Core extracts the useful information from the packet header/transaction and makes it available to the user in a more convenient format. For PCI Express, the core also strips the packet header and TLP digest (if present) and passes just the packet payload onto the Target data ports. For all cores, the packet/transaction data is shifted according to the starting address and Core Data Width and the associated byte enables are generated.

Received read requests (and associated read data) that map to BAR1-5 are completed using the Target Read Interface (targ_rd_* ports). For received target read requests, the DMA Back-End Core formulates the required completion packets/transactions on behalf of the user using the provided read data. For PCI Express, larger read requests which cross a 128 byte address boundary, are transparently broken into multiple Target Read Interface transactions to reduce overall read latency and to ensure that completions are not transmitted that exceed the system-allowed maximum payload size. Each PCI Express Target Read Interface transaction results in the transmission of one completion packet.

The Target Write Interface and Target Read Interface may be wait-stated by the user on a per transaction basis through the use of the targ_wr_user_ready and targ_rd_user_ready ports. Once a transaction has been granted, data must be provided/accepted when requested.

For applications where simplified data ordering rather than high throughput is more desirable or target write and target read coherency is required, the Target Write Interface and Target Read Interface can be combined into one unified interface by blocking the inactive interface from accepting transactions while the active interface completes a current transaction. Transaction blocking (wait-stating) is done through the use of the targ_wr_user_ready and targ_rd_user_ready ports and is generally desirable for low bandwidth control applications where it is desired to simplify transaction ordering and high throughput is unnecessary.

Target Write Interface and Target Reads Interface transactions only occur for PCI Express packets which decode to BAR1-5. All completions generated by the DMA Back-End Core are returned with status successful and must receive data from the user as requested. Transactions which fail to map to user resources are terminated by the PCI Express Core and are not handled by the DMA Back-End Core.

The Target Write Interface and Target Read Interface have a command phase in which transactions are granted and a data phase in which transaction data is transferred. The command and data phases are disassociated in time and overlapped to ensure that high throughput can be achieved on the interfaces. For example, a transaction is granted when there is a request and both the core and user indicate that they are ready to perform the transfer. Once granted, the core begins processing the packet header, formats the user data to the correct alignment, and implements the data transfer on the data ports of the associated interface. While the current transaction is in process, an additional transaction can be granted so that when the data phase of the current transaction completes, the next transaction's data phases can occur back-to-back. Note that for the PCI Express Core and DMA Back-End Core combination, packet header and framing is stripped/appended from/to the packet before/after the data transfer on the Target Interface, in many cases the transactions are being received or transmitted on PCI Express back-to-back, but will have apparent gaps on the target interfaces.

It is important to consume target transactions relatively quickly as it is possible to stall other receive traffic if target transactions are allowed to languish in the receiver buffer.

The Target Interface ports are listed in Table 6-1.

Table 6-1 Target Write Interface Ports

PORT	TYPE	DESCRIPTION
targ_wr_req	Output	Set when there is a target write transaction pending that targets a user resource. When targ_wr_req is asserted, targ_wr_cs indicates the target resource that the transaction is targeting.
targ_wr_core_ready	Output	Asserted when the core is ready to transfer a pending write transaction to the user. A transaction transfer is granted when targ_wr_req == 1, targ_wr_core_ready == 1, and targ_wr_user_ready == 1. Either the core or user can delay a transaction start by de-asserting their ready. The core will de-assert targ_wr_core_ready when the receive bus is being used to receive a different type of packet (such as a DMA completion, etc.). There will be a variable delay between a packet grant and the first data being available as the core consumes the header and formats the data and byte enables for the user. Once a transaction has been granted it is transferred with fixed timing and must be accepted. A second grant can be accepted while the data from a previous grant is still transferring so that the write data can be received at maximum rate.
targ_wr_user_ready	Input	Asserted when the user is ready to accept a pending write transaction. A transaction transfer is granted when targ_wr_req == 1, targ_wr_core_ready == 1, and targ_wr_user_ready == 1. The user must assert targ_wr_user_ready within a reasonable amount of time from targ_wr_req assertion to keep from stalling the PCI Express link/bus. Note that for PCI Express, the core does not start unpacking the target write request header until the transaction has been granted so cannot provide the address and length of the transaction until the first data phase. The user must, thus, be willing to accept a max payload size packet before asserting targ_wr_user_ready. Note that targ_wr_cs is valid with targ_wr_req and indicates which type of resource was targeted which can be useful in formulating targ_wr_user_ready. See targ_wr_core_ready above for additional information.
targ_wr_cs[5:0]	Output	Base Address Register which was hit for the current target write transaction: • 000001 – BAR1 • 000010 – BAR2 • 000100 - BAR3 • 001000 – BAR4 • 010000 – BAR5 • 100000 – Expansion ROM targ_wr_cs is only valid when targ_wr_req == 1. Note that only base address registers that are implemented by the PCI Express Core can be hit. Note that BAR0 is reserved for DMA and User Registers. Hits to BAR0 are handled by the DMA Back-End Core and do not appear on the Target Interface. Any hits to BAR0 that target user registers are handled by the user using the Register Interface.

Table 6-1 Target Write Interface Ports (Continued)

PORT	TYPE	DESCRIPTION
targ_wr_start	Output	Asserted when the data transfer for a previously granted write transaction begins. targ_wr_start is asserted the first clock cycle that targ_wr_en is asserted for each transaction.
targ_wr_addr[31:0]	Output	targ_wr_addr is valid when targ_wr_en is asserted and indicates to which address targ_wr_data should be written. targ_wr_addr increments with each targ_wr_en assertion in the same transaction. targ_wr_addr is a targ_wr_data-width address, so for a 32-bit core implementation is a 32-bit, DWORD address and for a 64-bit core implementation is a 64-bit, double DWORD address. Address bits which exceed the target BAR-size are echoed from the packet's/transaction's PCI Express address field and should be ignored.
targ_wr_count[12:0]	Output	targ_wr_count is only valid when targ_wr_en == 1 and indicates the number of clock cycles of data remaining in the current transaction. targ_wr_count decrements by one for each targ_wr_en.
targ_wr_en	Output	Set to transfer one write data from the DMA Back-End Core to the user; once a packet has been granted, the user must accept targ_wr_en assertions and the associated data when it arrives. targ_wr_en will assert and de-assert as data is taken from the underlying PCI Express Core and passed to the user. User logic must not depend upon fixed targ_wr_en behavior, since in some cases targ_wr_en will be variably gapped, even though in the majority of cases it will be continuously asserted for the entire transaction. For PCI Express, the amount of time between the grant and the data phases depends upon the Core Data Width and the size of the received PCI Express header. User logic should not assume a fixed timing relationship. The amount of clock cycles between the grant and the data phases will be relatively small especially at wider core data widths.
targ_wr_data[D-1:0]	Output	When targ_wr_en is asserted, targ_wr_data contains the associated write data. The DMA Back-End Core supports 32-bit, 64-bit, and 128-bit Core Data Width and is configured to have the same Core Data Width as the associated PCI Express Core. Core Data Width depends on the number of lanes and the capabilities of the target device. targ_wr_data width "D" corresponds to the Core Data Width.
targ_wr_be[B-1:0]	Output	Write byte enables: (1) write byte; (0) do not write byte; when targ_wr_en == 1, targ_wr_data is conditioned by targ_wr_be before being applied to the destination memory. The DMA Back-End Core supports 32-bit, 64-bit, and 128-bit Core Data Width. targ_wr_be width "B" corresponds to the Core Data Width/8.

Table 6-2 Target Read Interface Ports

PORT	TYPE	DESCRIPTION
targ_rd_req	Output	Set when there is a target read transaction pending that targets a user resource. When targ_rd_req is asserted, targ_rd_cs, targ_rd_addr, and targ_rd_count indicate the target resource that the transaction is targeting and the length of the transaction. In addition targ_rd_start is asserted the first clock of each new targ_rd_req assertion. Note that this is different from the Target Write interface since for reads, the core has this information in advance and can provide it to the user. In order to reduce latency to the first read data and to comply with maximum payload size restrictions, the DMA Back-End Core splits target read requests on 128-byte address boundaries and transparently handles this for the user. Thus a 512 byte read request starting on a 128-byte address boundary will result in 4 targ_rd_req assertions transferring 128-bytes each.
targ_rd_core_ready	Output	Asserted when the DMA Back-End Core needs data from the user to complete a pending read transaction. A transaction transfer is granted when targ_rd_req == 1, targ_rd_core_ready == 1, and targ_rd_user_ready == 1. Either the core or user can delay a transaction start by de-asserting their ready. The core will de-assert targ_rd_core_ready when the transmit bus is being used to transmit a different type of packet (such as a DMA write request, etc.). There will be a variable delay between a packet grant and the first data phase as the core adds the completion header to the packet and formats the data payload for the user. Once a transaction has been granted it is transferred with fixed timing and read data must be provided when requested. A second grant can be accepted while the data from a previous grant is still transferring so that the read data can be transmitted at maximum rate.
targ_rd_user_ready	Input	Asserted when the user is ready to provide data for the pending read transaction. A transaction transfer is granted when targ_rd_req == 1, targ_rd_core_ready == 1, and targ_rd_user_ready == 1.

Table 6-2 Target Read Interface Ports (Continued)

PORT	TYPE	DESCRIPTION
targ_rd_cs[5:0]	Output	Base Address Register which was hit for the current target read transaction: • 000001 – BAR1 • 000010 – BAR2 • 000100 – BAR3 • 001000 – BAR4 • 010000 – BAR5 • 100000 – Expansion ROM targ_rd_cs is valid whenever targ_rd_req is asserted. Note that only base address registers that are implemented by the PCI Express Core can be hit. Note that BAR0 is reserved for DMA and User Registers. Hits to BAR0 are handled by the DMA Back-End Core and do not appear on the Target Interface. Any hits to BAR0 that target user registers are handled by the user using the Register Interface.

Table 6-2 Target Read Interface Ports (Continued)

PORT	TYPE	DESCRIPTION
targ_rd_start	Output	Asserted the first clock cycle that a target read transaction becomes pending via targ_rd_req == 1. Note that this timing differs from the corresponding write port targ_wr_start. When targ_rd_start is asserted, targ_rd_cs, targ_rd_addr, and targ_rd_count all are valid for the new transaction. targ_rd_start is intended to be used by the user to begin fetching read data so that it can later assert targ_rd_user_ready when the data is ready.
targ_rd_addr[31:0]	Output	targ_rd_addr is only valid when targ_rd_req == 1 and indicates the starting read address for the pending read transaction. targ_rd_addr is a targ_rd_data-width address, so for a 32-bit core implementation is a 32-bit, DWORD address, for a 64-bit core implementation is a 64-bit, double DWORD address, etc. Address bits which exceed the target BAR-size are echoed from the packet's/transaction's PCI Express address field and should be ignored.
targ_rd_first_be[B-1:0]	Output	targ_rd_first_be is only valid when targ_rd_req == 1 and indicates the read byte enables for the first word of the pending read transaction. (1) read byte; (0) do not read byte. targ_rd_first_be width "B" corresponds to the Core Data Width/8.
targ_rd_last_be[B-1:0]	Output	targ_rd_last_be is only valid when targ_rd_req == 1 and indicates the read byte enables for the last word of the pending read transaction. (1) read byte; (0) do not read byte. targ_rd_last_be width "B" corresponds to the Core Data Width/8.
targ_rd_count[12:0]	Output	targ_rd_count is only valid when targ_rd_req == 1 and indicates the number of clock cycles of data requesting to be transferred in the current transaction.
targ_rd_en	Output	Asserted 1 clock before read data must be provided for a previously granted request. Once a read request is granted data must be provided when requested. targ_rd_en will assert and de-assert as data is needed to provide to the underlying PCI Express Core. User logic must not depend upon fixed targ_rd_en behavior, since in some cases targ_rd_en will be variably gapped, even though in the majority of cases it will be continuously asserted for the entire transaction. The amount of time between a read transaction grant and the read data phases is variable. User logic should not assume a fixed timing relationship. However, the amount of clock cycles between the grant and the data phases will be relatively small especially at wider core data widths.

Table 6-2 Target Read Interface Ports (Continued)

PORT	TYPE	DESCRIPTION
targ_rd_data[D-1:0]	Input	targ_rd_data must be valid one clock after a targ_rd_en assertion to transfer the read data of a previously granted request. The DMA Back-End Core supports 32-bit, 64-bit, and 128-bit Core Data Width and is configured to have the same core data width as the associated PCI Express Core. Core Data Width depends on the number of lanes and the target device. targ_rd_data width "D" corresponds to the Core Data Width. Note that there are not read byte enables for the target Read Interface, so all read data bytes should be provided for the entire targ_rd_count. Only bytes which were requested in the transaction will transfer on PCI Express and any remainder starting and ending bytes which were not requested will be discarded.

6.2 Target Interface Example Transactions

Figure 6-1 illustrates the timing of the Target Write Interface.

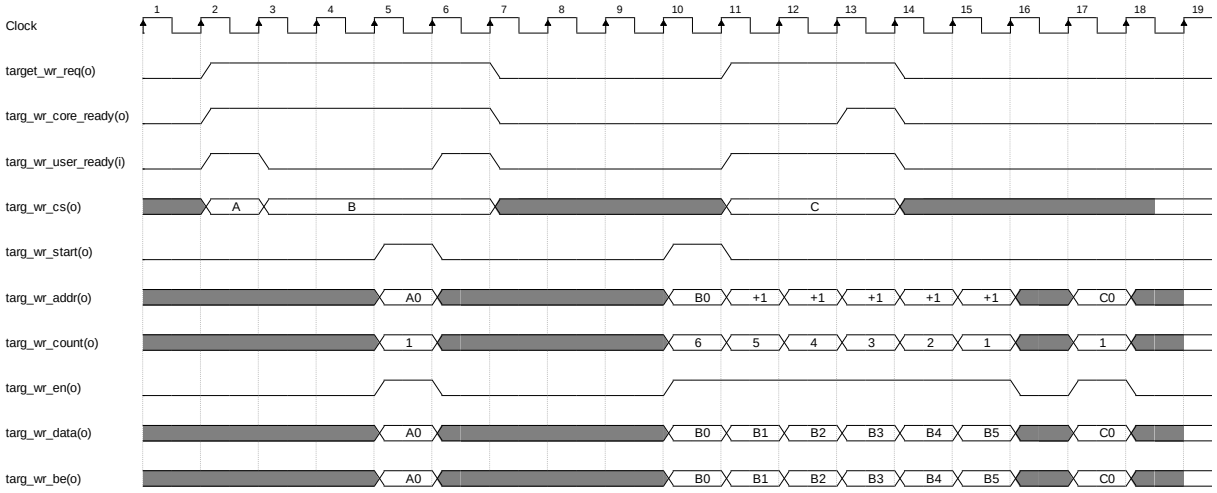


Figure 6-1 Target Write Interface Example Transactions

Transaction A: On cycle 1 targ_wr_req asserts indicating that the core has a target write pending for transfer to user resources and targ_wr_cs indicates which base address region the pending write is targeting. The core and user are both ready to begin receiving this transaction and assert targ_wr_core_ready and targ_wr_user_ready on cycle 1. targ_wr_req == targ_wr_core_ready == targ_wr_user_ready == 1 grants the transaction. The data phase of transaction A begins and ends (since it's a single word transaction) on cycle 4 with the assertion of targ_wr_start and targ_wr_en. Coincident with targ_wr_en on cycle 4, targ_wr_addr is the address to which the write should be written, targ_wr_count == 1 indicating it's a single word transaction, and targ_wr_data and targ_wr_be contain the data and bytes enables.

Transaction B: After Transaction A is granted on cycle 1, Transaction B can be seen to become pending on cycle 2 with the continued assertion of targ_wr_req. targ_wr_cs changes on cycle 2 with the new transaction to indicate the base address resource targeted by Transaction B. It is assumed in this case that the user does not want the complication of overlapping transactions, and decides to wait state the transaction until cycle 5 when it asserts targ_wr_user_ready. The data phases follow a few clocks later on cycles 9 through 14. On cycle 9, targ_wr_start asserts indicating the beginning of the data phases, targ_wr_addr indicates the starting address for the transaction, and targ_wr_count is 6 indicating that the transaction will transfer 6 words. On cycles 10 through 14, targ_wr_addr increments and targ_wr_count decrements with each additional targ_wr_en transfer.

Transaction C: Transaction C becomes pending on cycle 10 with the assertion of targ_wr_req. targ_wr_cs indicates the base address region that is being targeted for Transaction C. On cycles 10 and 11, it is assumed that the core is still receiving Transaction B and the core de-asserts targ_wr_core_ready until cycle 12, when it can transfer Transaction C back-to-back with Transaction B. For PCI Express Core's, the core must remove the PCI Express header from write packet before providing the data on the target interface, a gap in targ_wr_en is assumed to occur on cycle 15. The data phase of transaction C starts and ends on cycle 16 with the assertion of targ_wr_start, targ_wr_en, targ_wr_addr, targ_wr_count == 1, targ_wr_data, and targ_wr_be.

Note that the amount of time between a packet grant and the start of the data phases will vary to some degree and user logic should not count on a specific time delta between a grant and the associated data phases. Also, for PCI Express, smaller Core Data Width cores will have larger delays between grants and data phases since it will take more time for the core to strip the packet header.

Figure 6-2 illustrates the timing of the Target Read Interface.

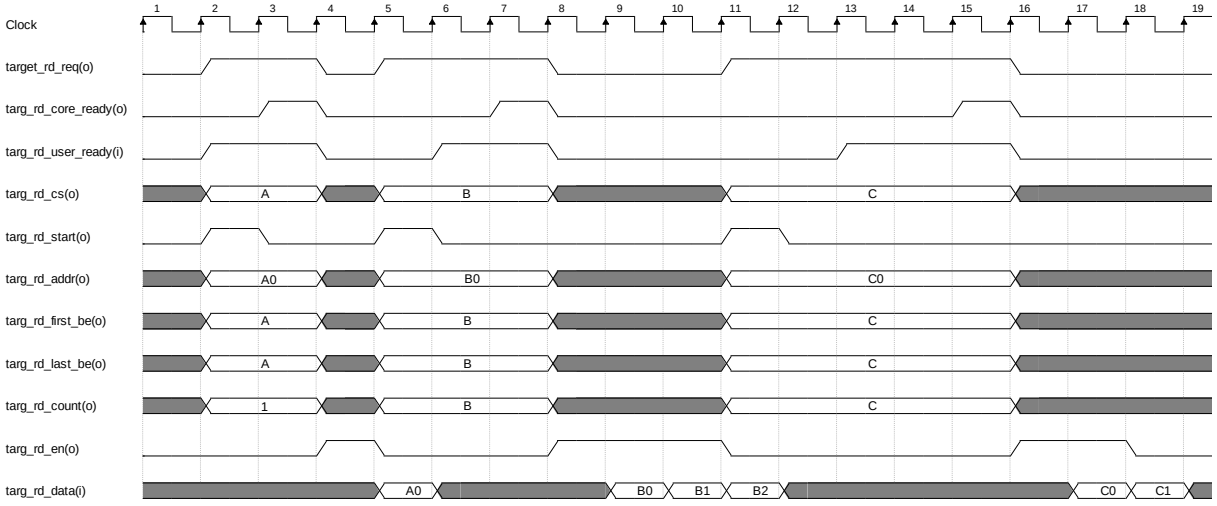


Figure 6-2 Target Read Interface Example Transactions

Transaction A: On cycle 1 targ_rd_req asserts indicating that the core has a target read pending to a user resource. On the same clock cycle, targ_rd_start asserts indicating the beginning of a request while targ_rd_cs indicates which base address region the pending read is targeting, targ_rd_addr indicates the transaction read starting address, targ_rd_first_be and targ_rd_last_be indicate the bytes that are being requested, and targ_rd_count indicates the number of words which are being requested. The user asserts targ_rd_user_ready on cycle 1 indicating that the user is ready to provide the requested data. The core begins arbitrating for the PCI Express bus upon seeing targ_rd_user_ready asserted on cycle 1 and when successful indicates that the transaction is granted by asserting targ_rd_core_ready on cycle 2. The data phase of Transaction A occurs with the targ_rd_en assertion on cycle 3 and targ_rd_data valid on cycle 4.

Transaction B: On cycle 4, Transaction B begins to be pending as indicated by targ_rd_req and targ_rd_start asserting. targ_rd_cs, targ_rd_addr, targ_rd-first_be, targ_rd_last_be, and targ_rd_count contain the transaction command information. In this case, the user logic is assumed to need one clock to become ready and asserts targ_rd_user_ready on cycle 5. On cycle 6, the core has secured access to the PCI Express bus and asserts targ_rd_core_ready to begin the transaction. The data phases occur with targ_rd_en asserting on cycles 7 through 9 and read data transferring on cycles 8 through 10.

Transaction C: On cycle 10, Transaction C begins to be pending as indicated by targ_rd_req and targ_rd_start asserting. targ_rd_cs, targ_rd_addr, targ_rd-first_be, targ_rd_last_be, and targ_rd_count contain the transaction command information. In this case, the user logic is assumed to need two clocks to become ready and asserts targ_rd_user_ready on cycle 12. It is assumed that the core requires a couple of clocks to secure access to the PCI Express bus and asserts targ_rd_core_ready on cycle 14. The data phases occur with targ_rd_en asserting on cycles 15 and 16 and read data transferring on cycles 16 and 17.

Note that the amount of time between a packet grant and the start of the data phases (targ_rd_en assertion) will vary to some degree and user logic should not count on a specific time delta between a grant and the associated data phases. For PCI Express, smaller Core Data Width cores will have larger delays between grants and data phases since the core will not begin to request read data from the user until after it has written the packet header.

7 Register Interface

The DMA Back-End Core implements a 64 Kbyte, 32-bit address Base Address Register using PCI Express Configuration Register locations 0x13-0x10. This region is used by the DMA Back-End Core to implement DMA and user registers.

The DMA Back-End Core reserves the lower half of the Base Address Register for current and future DMA registers. The remaining upper half of register space is available for user registers.

The Register Interface has fixed timing. All writes must be accepted at the rate that they occur and read data must be provided with fixed timing.

All register accesses appear on the register Interface, including the DMA Back-End Core register accesses, so the user must do a full address decode to determine if a user register is being accessed.

Use of this interface is optional. If unused, reg_rd_data must be tied to 0x0.

7.1 Register Map

Each DMA Engine implements a 256 byte register block, as described in Section 8.5, in the Register Base Address region. System to Card DMA Engines are located every 256 bytes starting at address offset 0x0. Card to System DMA Engines are located every 256 bytes starting at address offset 0x2000. This mapping is illustrated in **Table 7-1**.

Software can determine which DMA Engines are implemented by checking whether the DMA_Engine_Capabilities : Present bit is asserted at each of the possible locations. It is recommended that software be designed to check all 64 locations to support maximum flexibility in the mapping of engines.

Table 7-1 BAR0 Register Map

Byte Address	Type
0x00FF – 0x0000	System to Card Descriptor Engine[0] Register Block
0x01FF - 0x0100	System to Card Descriptor Engine[1] Register Block
0x02FF - 0x0200	System to Card Descriptor Engine[2] Register Block
0x03FF - 0x0300	System to Card Descriptor Engine[3] Register Block
0x1FFF - 0x0400	Reserved for additional System to Card DMA Engines
0x20FF – 0x2000	Card to System Descriptor Engine[0] Register Block
0x21FF - 0x2100	Card to System Descriptor Engine[1] Register Block
0x22FF - 0x2200	Card to System Descriptor Engine[2] Register Block
0x23FF - 0x2300	Card to System Descriptor Engine[3] Register Block
0x3FFF - 0x2400	Reserved for additional Card to System DMA Engines
0x40FF - 0x4000	DMA Common Register Block
0x7FFF - 0x4100	Reserved for DMA Back-End Core Future Use
0xFFFF - 0x8000	Available for user Registers implemented via the Register Interface

7.2 Register Interface Port Descriptions

The DMA Back-End Core Register Interface ports are described in Table 7-2.

Table 7-2 Register Interface Port Descriptions

PORT	TYPE	DESCRIPTION
reg_wr_addr[R-1:0]	Output	Address of the register being accessed in the current write transaction. reg_wr_addr is a reg_wr_data-width address into the 64 Kbyte BAR0 region. The number of address bits varies according to Core Data Width: • R == 12 bits for 128-bit Core Data Width • R == 13 bits for 64-bit Core Data Width • R == 14 bits for 32-bit Core Data Width reg_wr_addr increments by 1 for each transfer in a burst write operation.
reg_wr_en	Output	When reg_wr_en == 1, the register addressed by reg_wr_addr must be written with reg_wr_data conditioned by reg_wr_ be byte enables.
reg_wr_be[B-1:0]	Output	Active high byte enables; width B is the Core Data Width/8
reg_wr_data[D-1:0]	Output	Register write data; must be conditionally applied using the reg_wr_be byte enables; width D is the Core Data Width
reg_rd_addr[R-1:0]	Output	Address of the register being accessed in the current read transaction. reg_rd_addr is a reg_rd_data-width address into the 64 Kbyte BAR0 region. The number of address bits varies according to Core Data Width: • R == 12 bits for 128-bit Core Data Width • R == 13 bits for 64-bit Core Data Width • R == 14 bits for 32-bit Core Data Width reg_rd_addr increments by 1 for each transfer in a burst read operation.
reg_rd_data[D-1:0]	Input	Contents of the register addressed by reg_rd_addr. reg_rd_data must contain valid read data one clock after reg_rd_addr. If reg_rd_addr does not correspond to an implemented register, then reg_rd_data must be 0x0. The upper-most reg_rd_addr bit is 1 for a user register access and 0 for a DMA Back-End Core register access (see Table 7-1). DMA Back-End Core register accesses are handled entirely by the core but the accesses still occur on the Register Interface so a full address decode is required. If the Register Interface is unused reg_rd_data must be 0x0. reg_rd_data must be the output of a register to keep from introducing a long timing path in the DMA Back End since the Register Interface is shared by several DMA Back End register modules. Width D is the Core Data Width

Although the user is responsible for carrying out writes and reads to their user-implemented registers, all other aspects of register write and read transactions are handled for the user. The DMA Back-End Core consumes register write and read requests mapping to the Register Interface and generates the appropriate completion packets for register reads using the provided register read data.

When a write request is received that maps to the Register Base Address region, the DMA Back-End Core consumes the request and forwards the write information contained in the request onto the reg_wr_addr, reg_wr_en, reg_wr_data, and reg_wr_be ports for the DMA/user registers to perform the write.

When a read request is received that maps to the Register Base Address region, the DMA Back-End Core consumes the request, forwards the read information contained in the request onto the reg_rd_addr port, and creates a read completion packet with the data provided on the reg_rd_data port.

Register writes and reads can occur on consecutive clocks if consecutive packets are received and the Register Interface also supports bursts.

7.3 Register Interface Example Transactions

Figure 7-1 illustrates the timing of the Register Interface.

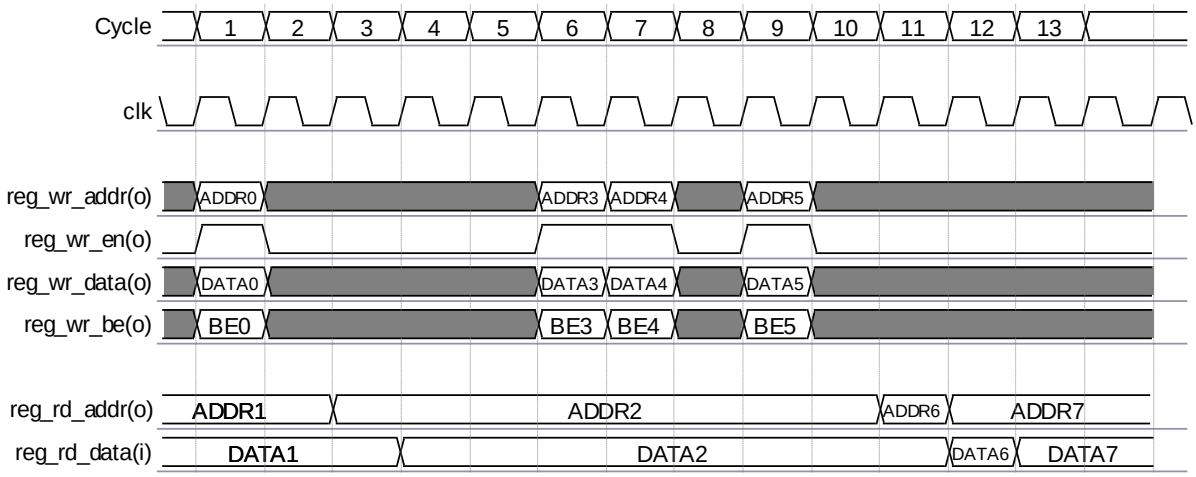


Figure 7-1 Register Interface Example Transactions

A register write occurs on the Register Interface on cycle 1, 6, 7, and 9 as indicated by reg_wr_en == 1. If reg_wr_addr on the appropriate cycle targets a user register, then the write must be applied to the user register located at reg_wr_addr using the data on reg_wr_data conditioned by the byte enables on reg_wr_be .

reg_wr_en assertions on cycle 6-9 illustrate a burst register write operation with a pause on cycle 8. Note that it is not necessary or possible to know whether the reg_wr_en on cycles 6-9 are part of one PCI Express burst write or adjacent singe register writes.

Register read data must always follow reg_rd_addr by 1 clock. This timing is illustrated by the reg_rd_data updating on cycles 4, 12, and 13.

reg_rd_addr changes on cycles 11 and 12 illustrates a read burst of 2. Note that it is not necessary or possible to know whether the reg_rd_addr changes are part of one PCI Express burst read or adjacent singe register reads.

8 DMA Interface

8.1 DMA Engine Operation

The DMA Back-End Core implements a high performance, multi-engine, scatter-gather, demand-driven DMA Implementation to enable high-throughput data transfer data between PCI Express and user logic.

Key features include:

- Very flexible, easy-to-use, high-performance DMA implementation
- Card-to-System (C2S) DMA Engine
  - Takes data from user logic and makes DMA Write Requests to system memory
  - Demand-driven user interface; user can wait state for flow control
  - Flexible Control - DMA Descriptor Engine fetches DMA Descriptors from a linked list of Descriptors stored in system memory or user logic can directly control the DMA Engine
- System-to-Card (S2C) DMA Engine
  - Makes DMA Read Requests from system memory, handles the resulting Read Completions, and forwards read data to user logic
  - Demand-driven user interface; user can wait state for flow control
  - Guarantees read data ordering (re-orders completions that were received out of order)
  - Flexible Control - DMA Descriptor Engine fetches DMA Descriptors from a linked list of Descriptors stored in system memory or user logic can directly control the DMA Engine
- High-performance multi-engine DMA with two software interface options
  - Packet DMA
    - Optimized for small and large transfers
    - Built-in support for packet applications
    - Built-in support for addressed card memory & FIFO applications
    - Enhanced functionality over Block DMA; recommended for new designs
  - Block DMA – Not recommended for new designs; Packet DMA implements a superset of Block DMA functionality; Block DMA is being maintained for existing customers
    - Optimized for large transfers
    - Built-in support for addressed card memory & FIFO applications
    - Software compatible with previous DMA Back-End “Gen3” versions
    - Mostly user-interface compatible with previous DMA Back-End “Gen3” versions; see Section 14 for migration details.
- Base Configuration has 1 Card to System DMA Engine and 1 System to Card DMA Engine
- Multi-Engine Configuration has 1-4 Card to System DMA Engines and 1-4 System to Card DMA Engines
- DMA Engines are fully independent and are interleaved on a packet basis
- 32-bit and 64-bit System Address Support
- 32-bit Descriptor Linked-List System Address Support
- 64-bit Card Address Support (Block DMA); 36-bit (Packet DMA)
- Option for user hardware design to provide Descriptors directly
- MSI and Legacy Interrupt support
- System address, Card Address, and transfer Byte Count fully support byte alignments allowing for maximum software flexibility
- Scatter-gather (chaining) DMA engines achieve high throughput even with heavily fragmented system memory
- Supports extremely long DMA transfers

For larger DMA transfer sizes, DMA operations increase throughput performance 10-100 times over conventional CPU-driven reads and writes. This performance gain is achieved by the DMA Engine through the use of overlapping read transactions and through large write and read burst sizes. A CPU is typically limited to generating very small (DWORD-size) writes and reads or very small bursts if the access is in/to a cached region.

The DMA Engines can be controlled from two different interfaces:

- Each DMA Engine has an associated DMA Register set which controls the fetching of DMA Descriptors from a linked-list of Descriptors stored in system memory and forwards them to the DMA Engine for execution. The DMA Registers are controlled via PCI Express target writes (from a software driver) to the DMA Engine's register block.
- User logic can provide their own Descriptors via the DMA Direct Control Interface. This is advantageous for applications where it is desired to have a local process or state machine control the DMA Engine rather than software entering though the PCI Express bus. Note that the user design must know where in system memory DMA data can be directed so that user logic can build the Descriptors. This requires the user design to communicate, at least minimally, with user software to obtain this information.
- The DMA Engine may be controlled either from the DMA Register set or through the DMA Direct Control interface, but not both simultaneously (there is no arbitration done between interfaces); all outstanding DMAs for the engine must fully complete before the control interface may be changed.

8.2 DMA Descriptor Definition

In a typical DMA application, application software wants to transfer a large amount of data between a buffer in system memory and a buffer in a PCI Express device. In order for the DMA Engine to access the application's buffer, the application buffer must be mapped from a virtual memory address into physical system memory addresses. The application buffer is normally heavily fragmented in physical memory and the operating system will typically map the application buffer into numerous, operating system page size memory fragments (page size for most operating systems is 4Kbytes). For example, if the application's buffer is 1 Megabyte, the Operating System may map this buffer in physical memory as 256 memory fragments of 4Kbytes each. To accomplish the desired 1 MByte transfer in this example, 256 DMA operations are required each with a different system starting address, and potentially, a different transfer byte count. In addition the PCI Express device buffer memory (Card Memory) may be similarly fragmented requiring that the DMA operation is further divided.

The information to describe the operation to transfer one contiguous memory fragment from/to system memory and card memory is called a Descriptor. A Descriptor may describe a very small memory fragment or a huge chunk of contiguous memory.

Note that for the purposes of this discussion, "system memory" refers to memory which is accessed through PCI Express using the memory mapped mechanism. This is most often the system's main SDRAM memory, but may also be memory on another PCI Express device in the system. Similarly "card memory" refers to memory that is accessed via the user DMA System to Card Interface and DMA Card to System Interfaces. This memory can be a FIFO in which case Card Address is meaningless, and can be ignored, or can be an addressable memory such as a local SDRAM module accessed through the user's design.

The DMA Engine allows a very large number of Descriptors to be simultaneously queued to minimize the impact of DMA Engine setup time on DMA throughput. This is accomplished by enabling the Descriptors to be stored in system memory and then fetched as needed.

Descriptors are 256-bits (32-bytes). Descriptors have different formats for Packet DMA and Block DMA.

8.2.1 Packet DMA Descriptor Format

A 256-bit (32-byte) Descriptor is defined for Packet DMA which contains the Control fields required to specify a packet copy operation and the Status fields required to specify the success/failure of the packet copy operation.

The Descriptor is split into Control and Status fields:

- Control fields are fields that are written into the Descriptor by software before the Descriptor is passed to the DMA Engine. Control fields specify to the DMA Engine what copy operation to perform.
- Status fields are fields that are written into the Descriptor by the DMA Engine after completing the DMA operation described in the Control portion of the Descriptor. Status fields indicate to software the Descriptor completion status. Software should zero all status fields prior to making the Descriptor available to the DMA Engine.
- To promote ease of re-using Descriptors (for circular queues), Control and Status fields are assigned their own locations in the Descriptor.

The Packet DMA Descriptor has the following format (addresses increment left and down):

- Card to System Direction (data flow is from card memory to system memory)
  - {C2SDescStatusFlags[7:0], Reserved[3:0], C2SDescByteCount[19:0]} - Status
  - C2SDescUserStatus[31:0] – Status
  - C2SDescUserStatus[63:32] – Status
  - DescCardAddr[31:0] – Control
  - {C2SDescControlFlags[7:0], DescCardAddr[35:32], DescByteCount[19:0]} - Control
  - DescSystemAddr[31:0] – Control
  - DescSystemAddr[63:32] – Control
  - {DescNextDescPtr[31:5], 5'b00000} – Control
- System to Card Direction(data flow is from system memory to card memory)
  - {S2CDescStatusFlags[7:0], S2CDescStatusErrorFlags[3:0], S2CDescByteCount[19:0]} – Status & Control
  - S2CDescUserControl[31:0] – Control
  - S2CDescUserControl[63:32] – Control
  - DescCardAddr[31:0] – Control
  - {S2CDescControlFlags[7:0], DescCardAddr[35:32], DescByteCount[19:0]} - Control
  - DescSystemAddr[31:0] – Control
  - DescSystemAddr[63:32] – Control
  - {DescNextDescPtr[31:5], 5'b00000} - Control

8.2.1.1 Card to System Descriptor Field Descriptions

Data flow for Card to System DMA is from the user design to system memory. The DMA Engine receives packets on its DMA Interface from the user hardware design and writes the packets into system memory at the locations specified by the Descriptors. The Packet DMA Engine assumes that the packet sizes are variable and unknown in advance. The Descriptor Status fields contain the necessary information for software to be able to determine the received packet size and which Descriptors contain the packet data. Packet start and end are indicated by the SOP and EOP C2SDescStatusFlag bits. A packet may span multiple Descriptors. SOP=1, EOP=0 is a packet start, SOP=EOP=0 is a packet continuation, SOP=0, EOP=1 is a packet end, and SOP=EOP=1 is a packet starting and ending in the same Descriptor. The received packet size is the sum of the C2SDescByteCount fields for all Descriptors that are part of a packet.

Descriptor fields specific to Card to System DMA:

- C2SDescControlFlags[7:0] – Control
  - Bit 7 – SOP – Set if this Descriptor contains the start of a packet; clear otherwise; only set for addressable Packet DMA
  - Bit 6 – EOP – Set if this Descriptor contains the end of a packet; clear otherwise; only set for addressable Packet DMA
  - Bits[5:2] – Reserved
  - Bit[1] – IRQOnError – Set to generate an interrupt when this Descriptor Completes with error; clear to not generate an interrupt when this Descriptor Completes with error
  - Bit[0] – IRQOnCompletion – Set to generate an interrupt when this Descriptor Completes without error; clear to not generate an interrupt when this Descriptor Completes without error
- C2SDescStatusFlags[7:0] – Status
  - Bit 7 – SOP – Set if this Descriptor contains the start of a packet; clear otherwise
  - Bit 6 – EOP – Set if this Descriptor contains the end of a packet; clear otherwise
  - Bits[5] – Reserved
  - Bit 4 – Error – Set when the Descriptor completes due to an error; clear otherwise
  - Bit 3 – C2SDescUserStatusHighIsZero – Set if C2SDescUserStatus[63:32] == 0; clear otherwise
  - Bit 2 – C2SDescUserStatusLowIsZero – Set if C2SDescUserStatus[31:0] == 0; clear otherwise
  - Bit 1 – Short – Set when the Descriptor completed with a byte count less than the requested byte count; clear otherwise; this is normal for C2S Packet DMA for packets containing EOP since only the portion of the final Descriptor required to hold the packet is used.
  - Bit 0 – Complete – Set when the Descriptor completes without an error; clear otherwise
- C2SDescByteCount[19:0] - Status
  - The number of bytes that the DMA Engine wrote into the Descriptor. If EOP=0, then C2SDescByteCount will be the same as the Descriptor size DescByteCount. If EOP=1 and the packet ended before filling the entire Descriptor, then C2SDescByteCount will be less than the Descriptor size DescByteCount. The received packet size is the sum of the C2SDescByteCount fields for all Descriptors that are part of a packet.
  - C2SDescByteCount is 20-bits so supports Descriptors up to 2^20-1 bytes. Note that since packets can span multiple Descriptors, packets may be significantly larger than the Descriptor size limit.
- C2SDescUserStatus[63:0] – Status
  - Contains application specific status received from the user when receiving the final data byte for the packet; C2SDescUserStatus is only valid if EOP is asserted in C2SDescStatusFlags. C2SDescUserStatus is not used by the DMA Engine and is purely for application specific needs to communicate information between the user hardware design and system software. Example usage includes communicating a hardware calculated packet CRC, communicating whether the packet is an Odd/Even video frame, etc. Use of C2SDescUserStatus is optional.
  - C2SDescUserStatusHighIsZero and C2SDescUserStatusLowIsZero are provided for ensuring coherency of status information. Please see Section 8.2.1.8 for details.

8.2.1.2 System to Card Descriptor Field Descriptions

Data flow for System to Card DMA is from system memory to the user design. Software places packets into the Descriptors and then passes the Descriptors to the DMA Engine for transmission. The DMA Engine reads the packets from system memory and provides them to the user hardware design on its DMA Interface. The software knows the packet sizes in advance and writes this information into the Descriptors. Software sets SOP and EOP S2CDescControlFlags during packet to Descriptor mapping to indicate Packet start and end information. A packet may span multiple Descriptors. SOP=1, EOP=0 is a packet start, SOP=EOP=0 is a packet continuation, SOP=0, EOP=1 is a packet end, and SOP=EOP=1 is a packet starting and ending in the same Descriptor. The transmitted packet size is the sum of all S2CDescByteCount fields for all Descriptors that are part of a packet. The Descriptor Status fields contain the necessary information for software to be able to determine which Descriptors the DMA Engine has completed.

Descriptor fields specific to System to Card DMA:

- S2CDescControlFlags[7:0] – Control
  - Bit 7 – SOP – Set if this Descriptor contains the start of a packet; clear otherwise
  - Bit 6 – EOP – Set if this Descriptor contains the end of a packet; clear otherwise
  - Bits[5:2] – Reserved
  - Bit[1] – IRQOnError – Set to generate an interrupt when this Descriptor Completes with error; clear to not generate an interrupt when this Descriptor Completes with error
  - Bit[0] – IRQOnCompletion – Set to generate an interrupt when this Descriptor Completes without error; clear to not generate an interrupt when this Descriptor Completes without error
- S2CDescStatusFlags[7:0] - Status
  - Bits[7:5] – Reserved
  - Bit 4 – Error – Set when the Descriptor completes due to an error; clear otherwise
  - Bits[3:2] - Reserved
  - Bit 1 – Short – Set when the Descriptor completed with a byte count less than the requested byte count; clear otherwise; this is generally an error for S2C Packet DMA since packets are normally not truncated by the user design.
  - Bit 0 – Complete – Set when the Descriptor completes without an error; clear otherwise
- S2CDescStatusErrorFlags[3:0] – Status – Additional information as to why S2CDescStatusFlags[4] == Error is set. If S2CDescStatusFlags[4] == Error is set then one or more of the following bits will be set to indicate the additional error source information.
  - Bit 3 – Reserved
  - Bi t2 – Set when received one or more DMA read data completions with ECRC Errors
  - Bit 1 – Set when received one or more DMA read data completions marked as Poisoned (EP == 1)
  - Bit 0 – Set when received one or more DMA read data completions with Unsuccessful Completion Status
- S2CDescUserControl[63:0] – Control
  - Contains application specific control information to pass from software to the user hardware design; the DMA Engine provides the value of S2CDescUserControl to the user design the same clock that SOP is provided. S2CDescUserControl is not used by the DMA Engine and is purely for application specific needs. Use of S2CDescUserControl is optional.
- S2CDescByteCount[19:0] – Control & Status
  - Control - During packet to Descriptor mapping, software writes the number of bytes that it wrote into the Descriptor into S2CDescByteCount. If EOP=0, then S2CDescByteCount must be the same as the Descriptor size DescByteCount. If EOP=1 and the packet ends before filling the entire Descriptor, then S2CDescByteCount is less than the Descriptor size DescByteCount. The transmitted packet size is the sum of the S2CDescByteCount fields for all Descriptors that are part of a packet
  - S2CDescByteCount is 20-bits so supports Descriptors up to 2^20-1 bytes. Note that since packets can span multiple Descriptors, packets may be significantly larger than the Descriptor size limit.
  - Status – After completing a DMA operation, the DMA Engine writes the number of bytes transferred for the Descriptor into S2CDescByteCount. Except for error conditions, S2CDescByteCount should be the same as originally provided.

8.2.1.3 Common Descriptor Field Descriptions

The following Descriptor fields are common to both directions of DMA:

- DescCardAddr[35:0] – Control
  - Card starting address for this Descriptor in card memory; the Card Address is provided to the user on the DMA Interface so that the data can be placed into card memory at the addressed location. Use of DescCardAddress is optional.
- DescByteCount[19:0] – Control
  - DescByteCount is the size of the Descriptor, as provided by the operating system, when the application buffer was mapped into physical memory.
- DescSystemAddr[63:0] – Control
  - DescSystemAddr is the system starting address for this Descriptor in system memory; used by the DMA Engine to get/put data from/to system memory. DescSystemAddr[63:32] must == 0 if the system address is a 32-bit address.
- DescNextDescPtr[31:5] – Control
  - 32-byte address aligned pointer to the next Descriptor in the chain or 0x0 if the current Descriptor is the last Descriptor in the Descriptor chain.

8.2.1.4 Descriptor Packet Support

Descriptors contain SOP and EOP fields to demarcate packets within a DMA stream.

For C2S Direction:

- The DMA Engine receives SOP, EOP, UserStatus, and packet data payload from the user design
- SOP, EOP and individual Descriptor length are used to place the user's packet into one or more consecutive Descriptors as required
- SOP status is set == 1 for Descriptors containing the first data byte of a packet.
- EOP status is set == 1 for Descriptors containing the last data byte of a packet
- Descriptors that contain neither the first nor last data byte of a packet are assigned SOP status == 0 and EOP status == 0 (a Descriptor in the middle of a packet)
- A Descriptor with SOP status == 1 and EOP status == 1 is the only Descriptor for this packet
- When using packet addressing, any descriptor with SOP status == 1, must provide a value for DescCardAddr to indicate the source address for the packet.
- When EOP is received from the user, the current Descriptor is truncated to the size required to hold the received packet data up to and including the data bytes provided the same clock that EOP was asserted.
- When a Descriptor is completed, or truncated by EOP, C2SDescByteCount status is updated with the amount of data actually transferred for the Descriptor; in the general case this will be the same value as the DescByteCount control field; however in the case of an EOP condition occurring before the Descriptor is fully utilized (for example if S/W does not know the packet size in advance) C2SDescByteCount status indicates the number of bytes that were used in the Descriptor.
- In the case of an EOP condition occurring before the Descriptor is fully utilized, the remaining portion of the Descriptor is unused. The next packet received will be placed starting with the start of the next Descriptor in the chain.
- UserStatus that is provided by the user design the same clock as EOP is copied into the C2SDescUserStatus status field of the Descriptor that contains the corresponding EOP == 1 status.
- The packet length is the sum of the C2SDescByteCount Descriptor status fields from the Descriptor with SOP status == 1 to the Descriptor with EOP status == 1 inclusive
- Descriptor completion status is written in one write operation to the Descriptor system memory after all DMA data associated with the Descriptor has been written

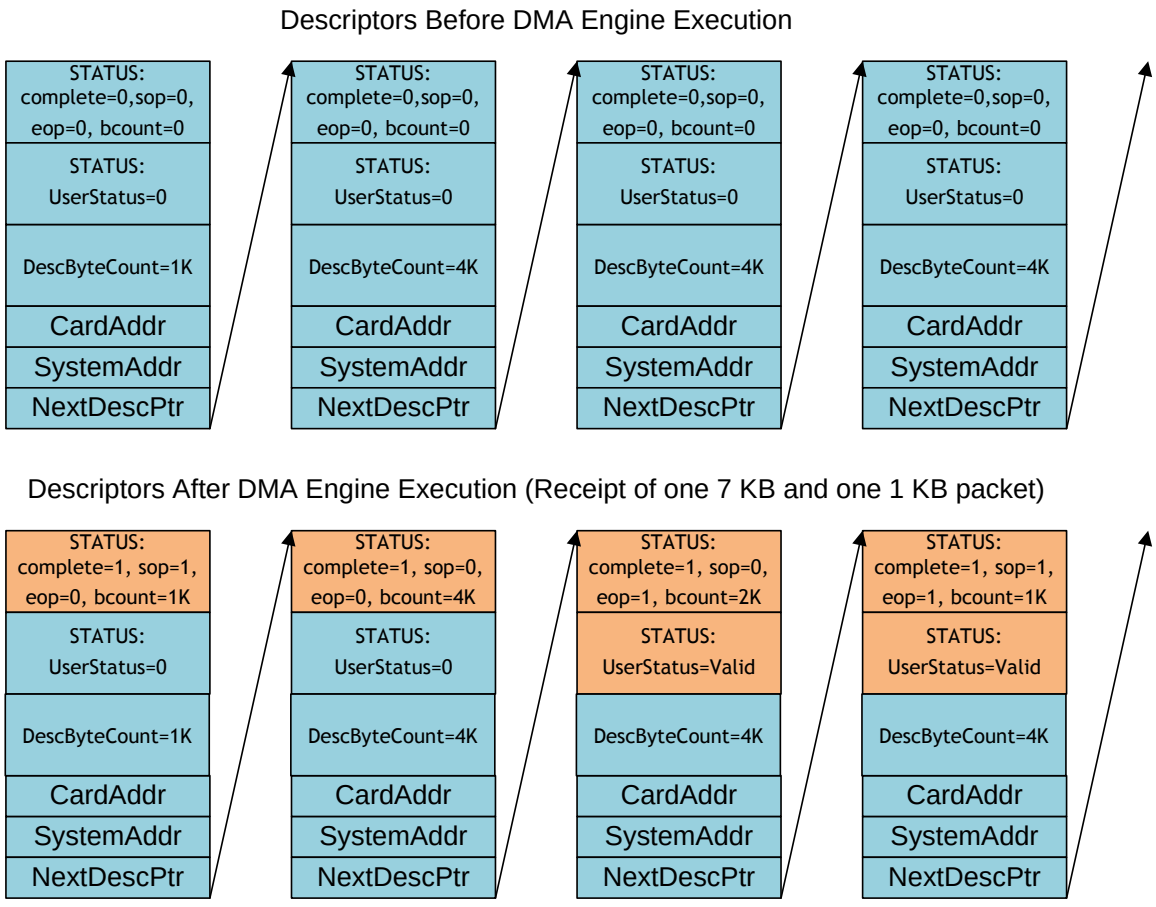
For S2C Direction:

- The DMA Engine uses the SOP and EOP S2CDescControl flags and the S2CDescByteCount control fields of consecutive Descriptors to re-generate packets and properly assert SOP and EOP for the user design
- SOP is set == 1 for Descriptors containing the first data byte of a packet
- EOP is set == 1 for Descriptors containing the last data byte of a packet
- Descriptors that contain neither the first nor last data byte of a packet are assigned SOP == 0 and EOP == 0 (a Descriptor in the middle of a packet)
- A Descriptor with SOP == 1 and EOP == 1 is the only Descriptor for this packet
- When using packet addressing, any descriptor with SOP status == 1, must provide a value for DescCardAddr to indicate the destination address for the packet.
- The packet length is the sum of the S2CDescByteCount control fields of the Descriptors from the Descriptor with SOP == 1 to the Descriptor with EOP == 1 inclusive
- S2CDescUserControl control information is passed to the user design the same clock that SOP is provided

8.2.1.5 Packet DMA Descriptor Examples

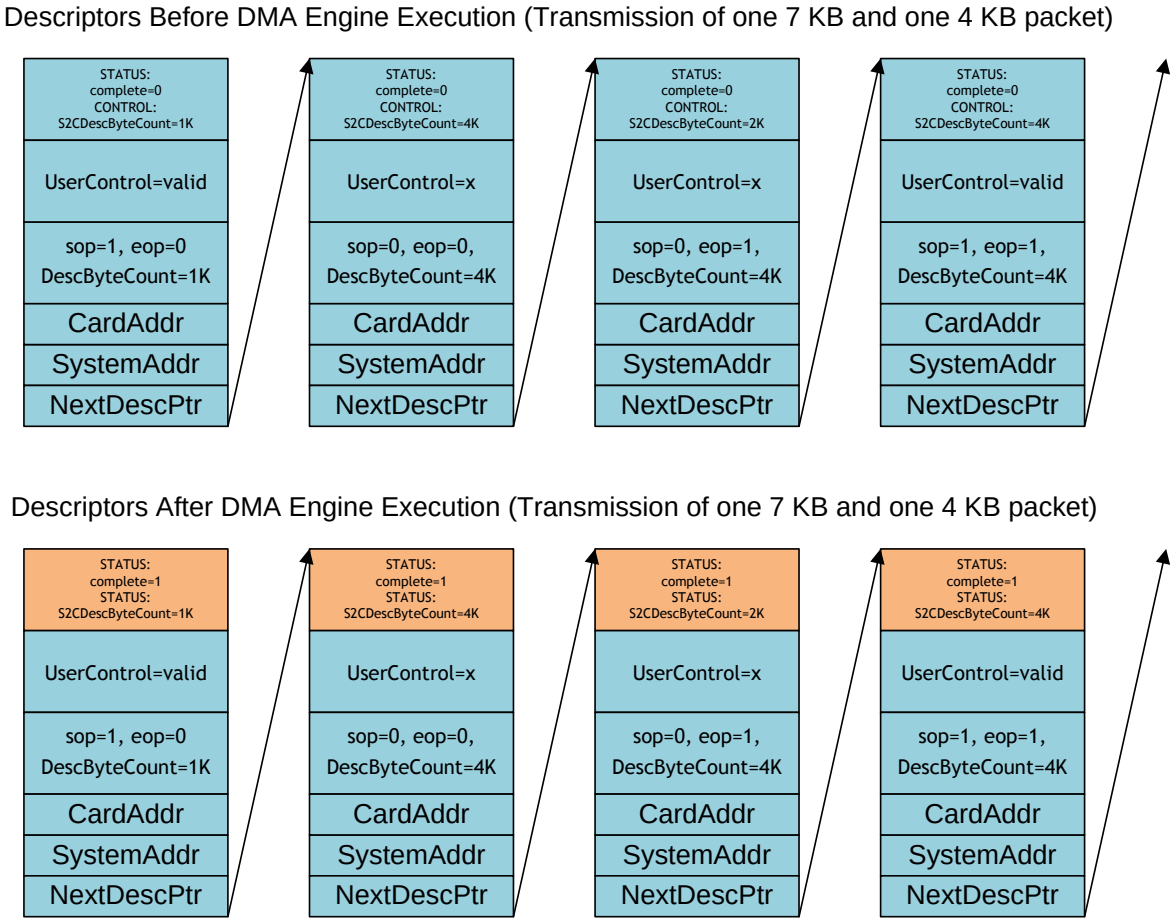
The following diagram illustrates the updating of Card to System DMA Descriptors in response to the DMA Engine having completed transfer of 2 packets. Note that the first packet spans 3 Descriptors and the second packet is contained in a single Descriptor. Note that in both cases it is assumed that the packets ended without consuming the entire final Descriptor. Note that UserStatus is only valid and updated for packets which contain eop=1. Modified Descriptor fields are shaded in orange.

8-1 Card to System Packet DMA Descriptor Example



The following diagram illustrates the updating of System to Card DMA Descriptors in response to the DMA Engine having completed transfer of 2 packets. Note that the first packet spans 3 Descriptors and the second packet is contained in a single Descriptor. Note that UserControl is only valid for packets which contain sop=1. Note that the first packet is 7KB while the 3 Descriptors which are used for the packet together are a total of 9KB. The second packet is 4KB and completely fills its 4KB Descriptor. Modified Descriptor fields are shaded in orange.

8-2 System to Card Packet DMA Descriptor Example



8.2.1.6 Descriptor Ownership

DMA Descriptors are generated by host system software and are fetched and executed by the DMA Engine. Since Descriptors are larger than the typical system atomic transaction size, it is important that a mechanism be defined such that each Descriptor is accessed only by software or only by the DMA Engine at any given time. Reg_Completed_Desc_Ptr and Reg_SW_Desc_Ptr define Descriptor ownership. Hardware owns the Descriptors between, but not including, Reg_Completed_Desc_Ptr (pointer to last completed Descriptor) and Reg_SW_Desc_Ptr (pointer to first Descriptor still owned by Software). Hardware advances Reg_Completed_Desc_Ptr to indicate that Descriptors have been completed and may be recycled and/or their data processed. Software advances Reg_SW_Desc_Ptr to give hardware additional Descriptors to execute.

8.2.1.7 Checking for Descriptor Completion

The Packet DMA Engine is designed to be flexible and allow for several different software packet-processing implementations.

There are two methods software can use to determine whether Descriptors that have been given to the DMA Engine to process have completed:

- Read Reg_Completed_Desc_Ptr
  - S/W reads the current value of the DMA Engine Reg_Completed_Desc_Ptr register to determine if any Descriptors have completed since the last time Reg_Completed_Desc_Ptr was read.
- Read the Descriptor Status Field of the next Descriptor to complete in the chain
  - S/W reads the status of the next Descriptor to complete in system memory. If the status indicates that the Descriptor completed, then software processes the Descriptor and then repeats the process for the next Descriptor.
- Descriptors are stored in system memory, so a read of the Descriptor status is very fast. Reads of the DMA Engine Reg_Completed_Desc_Ptr register need to make a round trip through the PCI Express hierarchy and thus are considerably slower.

The Packet DMA Engine allows for several methods to be used to determine when to check for Descriptors which have completed:

- The software driver implements a timer and periodically checks for new Descriptors when the timer expires. Any new Descriptors which have completed are processed. This process is simple to implement, and is a good choice if the timer interval is sufficiently long that this polling operation does not take significant system resources.
- An alternate of the above method, is for the software driver to only check for new packets when the application asks if there are packets to process. This effectively makes the application have to poll, which while extremely simple, will generally lead to less performance.
- Software enables the IRQ_On_Completion Descriptor control bit for one Descriptor every N Descriptors. When Descriptors complete which have IRQ_On_Completion set, software will receive an interrupt that can be used to check for new Descriptors to process. This method requires Descriptors to complete with regularity to be effective.
- An improvement of the above method is to the the above IRQ_On_Completion method for primary packet processing, but also to implement a timeout mechanism to process packets when the data path stalls or ends without an interrupt event.
- Software enables the IRQ_On_Completion Descriptor control bit for only the last Descriptor in every packet. Whenever the Descriptor at the end of the packet completes, software will receive an interrupt that can be used to check for new Descriptors to process.

8.2.1.8 Ensuring Coherency

Whichever method is used to determine Descriptor completion, it is important to ensure that DMA completion status and data are valid. A robust software implementation that maintains proper transaction coherency must implement a solution for the following complications:

- Memory writes and messages are allowed to pass other memory writes and messages in PCI Express (and in systems in general). Interrupt packets, DMA write data packets, and Descriptor status update packets are all write transactions, so it's possible for the ordering of write transactions to be modified en route to software such that software can register a DMA completion event before the DMA completion status or data is fully updated. A mechanism which may be used to ensure that all DMA Engine writes are flushed through to their destination is to do a PCI Express read of a DMA Engine register before using any DMA completion information received as part of a PCI Express write transaction. Since PCI Express ordering rules do not allow reads or read completions to pass writes, all writes occuring before a read must complete before the read will be allowed to complete.
- Systems in general do not guarantee transaction ordering between PCI Express transactions and system memory transactions since these transactions target different resources and are handled by distinct hardware in often different physical locations. Software gets some DMA information from PCI Express transactions (interrupts) and some information from system memory (DMA Descriptor Completion Status and DMA data). Thus the two paths should be considered asynchronous and no ordering for transactions targeting these different destinations should be assumed.
- Host platforms (Host CPU/Memory Controller implementation) normally only guarantee atomic operations of 32-bits (DWORD) for asynchronous operations, so it's possible that a greater than 32-bit DMA Engine Descriptor completion Status write to system memory could be interrupted in the middle by a Host Descriptor status read of the same locations. This creates the possibility that a DMA Completion Status read will get some new data and some old data.

The Packet DMA Engine and software follow the following process to ensure validity of DMA Descriptor completion status information:

- Software guarantees that all Descriptor status reads are done at multiples of 32-bit size and are 32-bit address aligned. Additionally software zeros all Descriptor Status fields before making the Descriptors available to the DMA Engine.
- To test for Descriptor completion, software reads the first 32-bits of the next Descriptor's Status. If C2SDescStatusFlags/S2CDescStatusFlags bits Complete == 1 or Error==1 then the Descriptor Completed and the 32-bits of status is valid. If Complete == 0 and Error == 0, then the Descriptor has not completed and the status is not valid. The location is read until valid status is obtained.
    - For System to Card DMA applications, this is enough to guarantee coherency since Descriptor status is always 32-bits
    - For Card to System DMA applications which do not use C2SDescUserStatus[63:0], this is enough to guarantee coherency since Descriptor status is always 32-bits
    - Card to System DMA applications which use any portion of C2SDescUserStatus[63:0] must implement an additional safeguard mechanism since the status is 33-bit to 96-bits which exceeds the typical system's 32-bit atomic size. To ensure that C2SDescUserStatus[63:0] is valid, the value of C2SDescUserStatusHighIsZero and C2SDescUserStatusLowIsZero are recorded when the first 32-bits of primary status were read and indicated Complete == 1 or Error == 1.
        - If C2SDescUserStatusLowIsZero == 1, then C2SDescUserStatus[31:0] is known to be 0 and does not need to be read. If C2SDescUserStatusLowIsZero == 0, then C2SDescUserStatus[31:0] is known to be non-zero and is read until a non-zero value is obtained indicating the value was updated by the DMA Engine and is valid.
        - If C2SDescUserStatusHighIsZero == 1, then C2SDescUserStatus[63:32] is known to be 0 and does not need to be read. If C2SDescUserStatusHighIsZero == 0, then C2SDescUserStatus[63:32] is known to be non-zero and is read until a non-zero value is obtained indicating the value was updated by the DMA Engine and is valid.
        - Note C2SDescUserStatus is only updated by the DMA Engine for Descriptors that contain the end of a packet. Thus it is not necessary to follow the above process to obtain C2SDescUserStatus for packets that do not have the EOP status bit set.

8.2.2 Block DMA Descriptor Format

A 256-bit (32-byte) Descriptor is defined for Block DMA which contains the Control fields required to specify a packet copy operation.

Table 8-1 Block DMA Descriptor Definition

Byte Address	Name
0x3 - 0x0	Control[31:0] • Bit[0] – Interrupt_on_Completion : If Interrupt_on_Completion == 1, then an interrupt will be generated when the DMA operation described by this Descriptor completes. Software can control the frequency of interrupts by setting this bit only for specific Descriptors. • Bit[1] – Interrupt on Error Short: Set to generate an interrupt when this Descriptor finishes short due to an error. • Bit[2] – Interrupt on S/W Short: Set to generate an interrupt when this Descriptor finishes short due to a software requested chain_stop. • Bit[3] – Interrupt on H/W Short: Set to generate an interrupt when this Descriptor finishes short due to a hardware (DMA user-interface) requested stop. • Bits[7:4] – Reserved. Set to 0000. • Bit[8] – Reserved. Was “Sequence” in prior DMA Back-End versions. This feature is no longer supported. • Bit[9] – Reserved. Was “Continue” in prior DMA Back-End versions. This feature is no longer supported. • Bit[10] – Last Descriptor in Chain: For the DMA Direct Control Interface only, when set causes c2s[]_data_last_chain to assert coincident with the last c2s[]_data_en of this Descriptor. When using the DMA Registers instead of DMA Direct Control, this bit is automatically managed by the register set and software does not control this bit. • Bits[31:11] – Reserved. These bits must be 0. Future enhancements may define additional bits for which 0 will not enable the new feature. The Card to System and System to Card DMA Engines support transfer counts and addresses that are in units of bytes, and thus, user-side DMA transfers will often start and end on addresses that are not aligned to the Back-End Core Data Width. DMA Chain status is typically only desired when an entire Descriptor Chain (Linked List of Descriptor) is completed. Thus it is recommended to set Interrupt on Completion only for the final Descriptor in a chain and to set Interrupt on Error Short, Interrupt on S/W Short, and Interrupt on H/W Short for every Descriptor in a chain. This combination ensures that one interrupt will be thrown for every chain whether it completes or ends short due to an error or stop request.

Table 8-1 DMA Descriptor Definition (Continued)

Byte Address	Name
0x7 - 0x4	Byte_Count[31:0] Number of bytes to transfer in this DMA operation. DMA Descriptor Byte Count supports between 1 and 2^32-1 bytes (1 Byte to 4GBytes-1) although it is common for smaller (4KByte to 1MByte) descriptors to be used. 0x0 is a reserved value and cannot be used. Bytes are written/read contiguously starting at System Starting Address on PCI Express and Card Starting Address on the Local Interface until Byte_Count is satisfied
0x0F - 0x8	System_Starting_Address[63:0] Starting address in system memory where the DMA data will be read/written. The upper 32-bits must be 0 if a 32-bit address is written into this register.
0x17 - 0x10	Card_Starting_Address[63:0] Starting address in card memory where the DMA data will be read/written. For user DMA applications targeting FIFOs which do not have a concept of a Card Address, Card_Starting_Address can be left unused or it can be re-tasked to provide different information for the user such as which of several FIFOs the data should access. Unused address bits should be written with 0.
0x1F - 0x18	Next_Descriptor_Pointer[63:0] Starting address in system memory where the next Descriptor is located or 0x0 if this is the last Descriptor in the linked list (chain). Descriptors must be aligned on Descriptor-size boundaries (32-bytes or 256-bits), so Next_Descriptor_Pointer[4:0] must be 00000. Forcing this alignment ensures that the Descriptor can be fetched in one read operation and simplifies the DMA Descriptor fetching logic resulting in fewer logic resources being required. The upper 32-bits must be 0 if a 32-bit address is written into this register.

8.3 DMA Interface Port Descriptions

The DMA Interface is the mechanism through which user logic interacts with the DMA Engine. The DMA Interface orchestrates the flow of DMA data between user logic and PCI Express.

The Card to System and System to Card interfaces are very similar. The main difference is the direction of data flow. Each interface is split into a set of command ports (cmd) which are used to control the flow of data and control information and a set of data ports (data) to carry out data transfer.

The command ports provide advance information/control, do not participate directly in data transfer, and are thus unnecessary for some applications.

The data ports control the flow of DMA data between the DMA Back-End Core and user logic. The user is in control of the data flow and can insert wait states between requests when necessary. Each data port request corresponds to a PCI Express packet/transaction (read request for S2C and write request for C2S).

After acceptance of a command on the command ports, the data is transferred on the data ports under user control. For Card to System user-interfaces, it is advantageous in many instances to use the command port information to begin fetching data for the DMA operation rather than using the individual Data Port requests due to the relatively small burst size in use on the data ports (typically 128-512 bytes). This is especially true for user applications with high read latencies.

There is one Card to System Interface for each Card to System DMA Engine and one System to Card Interface for each System to Card DMA Engine.

The DMA Interface ports are listed in Table 8-2, Table 8-3, Table 8-4, Table 8-5, Table 8-6, and Table 8-7. In the tables, size constant 'C' is the number of implemented Card to System Engines and size constant 'S' is the number of implemented System to Card Engines.

The DMA Interface ports are subdivided into three groups (Command, Data Request, Data Transfer) for Card to System and two groups (Command, Data Request/Transfer) for System to Card according to the relative timing of each portion of the interface. This division is intended to help clarify which signals are tightly coupled with one another.

In order to promote a wide variety of applications and to simplify user hardware designs, the DMA Interface provides a lot of informational/optional signals which are not needed by all applications. A typical design may only need to use half of the available DMA Interface signals.

Table 8-2 DMA Card to System Interface (Command)

PORT	TYPE	DESCRIPTION
c2s[C-1:0]_cmd_rst_n	Output	Active low DMA Engine reset. When 0 the DMA Engine is in reset and all user logic that interacts with the DMA Engine must be reset. Any outstanding DMA transactions will not complete.
c2s[C-1:0]_cmd_req	Output	Asserted by the DMA Engine to request a new Descriptor start. When c2s_cmd_req is asserted, c2s_cmd_addr, c2s_cmd_bcount, c2s_cmd_user_control, c2s_cmd_first_chain, and c2s_cmd_last_chain indicate the Card Starting Address, Byte Count, control word, and first/last Descriptor information for an entire Descriptor. In this direction of data flow it is advantageous for the user to know in advance what data is requested as part of the Descriptor. This allows the user design to prefetch data in advance of requests on the data ports in order to reduce data latency for increased throughput.
c2s[C-1:0]_cmd_ready	Input	The user asserts c2s_cmd_ready to signal acceptance of the Descriptor information indicated on c2s_cmd_req, c2s_cmd_addr, c2s_cmd_bcount, c2s_cmd_user_control, c2s_cmd_first_chain, and c2s_cmd_last_chain. This indicates that the user has saved/no longer needs this information and the DMA Engine can begin making data transfer request for this Descriptor. Use of the command portion of this interface is optional. If the command portion of the interface is unused, then c2s_cmd_ready must be tied to 1 and the remaining command ports ignored.
c2s[C-1:0]_cmd_addr[63:0]	Output	Card Starting Address for the DMA operation. Only valid when c2s_cmd_req == 1. c2s_cmd_addr is a byte address. This value is copied directly from the DMA Descriptor. For Block DMA, 64-bits are available. For Packet DMA 36-bits are available.
c2s[C-1:0]_cmd_bcount[31:0]	Output	Number of bytes to transfer in the current DMA operation. Only valid when c2s_cmd_req == 1. This value is copied directly from the DMA Descriptor.
c2s[C-1:0]_cmd_user_control[31:0]	Output	Descriptor User Control word. Only valid when c2s_cmd_req == 1. This value is copied directly from the DMA Descriptor. This field is not used for Card to System DMA and should be ignored.
c2s[C-1:0]_cmd_first_chain	Output	Set (1) if this Descriptor is the first Descriptor in a chain/packet. Clear (0) otherwise. Only valid when c2s_cmd_req == 1.
c2s[C-1:0]_cmd_last_chain	Output	Set (1) if this Descriptor is the last Descriptor in a chain/packet. Clear (0) otherwise. Only valid when c2s_cmd_req == 1.
c2s[C-1:0]_cmd_abort	Output	Asserted if a DMA operation is aborted by software request. All DMA transactions that are outstanding should be allowed to complete if possible. Then any remaining data that has been prefetched as part of a prior DMA operation should be flushed. Once the user design has terminated and cleaned-up following the DMA abort, the user logic must assert c2s_cmd_abort_ack for one clock indicating that the abort is acknowledged and the user logic is ready for a new DMA operation or DMA reset to occur.
c2s[C-1:0]_cmd_abort_ack	Input	See c2s_cmd_abort description for details.

Table 8-3 DMA Card to System Interface (Data Request)

PORT	TYPE	DESCRIPTION
c2s[C-1:0]_data_req	Output	Asserted by the DMA Engine to request a data transfer. When c2s_data_req is asserted, c2s_data_addr and c2s_data_bcount indicate the Card Starting Address and Byte Count for the requested transaction.
c2s[C-1:0]_data_ready	Input	The user asserts c2s_data_ready to grant permission to the DMA Engine to execute the data transfer indicated on the c2s_data_req, c2s_data_addr, and c2s_data_bcount ports. Once a transaction has been granted, the data for the transaction will follow on the c2s_data_en and c2s_data_data ports when the DMA Engine can secure the data resources to complete the transfer. The time from a granted request to the first data will vary and no assumptions should be made. This interface requires overlapping requests to achieve the best throughput performance. User logic should be designed to grant an additional request as early as possible and will need to be able to grant a new request while the data from the previously granted request is still being transferred to achieve maximum throughput.
c2s[C-1:0]_data_addr[63:0]	Output	Card Starting Address for the DMA operation. Only valid when c2s_data_req == 1. c2s_data_addr is a byte address. This value starts at the value contained in the DMA Descriptor and is incremented with each grant by the number of bytes which were granted. c2s_data_addr is not necessary for FIFO applications so can be unused or re-tasked to provide control information. For Block DMA, 64-bits are available. For Packet DMA 36-bits are available.
c2s[C-1:0]_data_bcount[9:0]	Output	Number of bytes to transfer in the current DMA operation. Only valid when c2s_data_req == 1. This port will typically be max payload size bytes (128, 256, or 512 bytes) but the beginning and ending transfers will vary according to the Descriptor System Starting Address offset and Byte Count.
c2s[C-1:0]_data_req_remain[R-1:0]	Output	Data remainder for the final data word of this request. Only valid when c2s_data_req == 1. Optional in many applications. R is the {4, 3, 2} when the core data width is {128, 64, 32} bits respectively.
c2s[C-1:0]_data_req_last_desc	Output	Set (1) when this is the last data_req for the current Descriptor. Clear (0) otherwise. Only valid when c2s_data_req == 1.
c2s[C-1:0]_data_stop	Input	Asserted instead of c2s_data_ready to request that the command be stopped before the end of the command. When c2s_data_stop is asserted, c2s_data_stop_bcount[9:0] must contain the number of additional bytes that the user desires to transfer beyond those which were already promised via previous c2s_data_ready assertions. c2s_data_stop_bcount[9:0] must have a value between 0 (no additional transfers) and c2s_data_bcount[9:0]. If c2s_data_stop_bcount is non-zero, then an additional c2s_data_req will occur with c2s_data_bcount == c2s_data_stop_bcount. c2s_data_stop functionality is provided for applications such as networking where the data transfer sizes are not known in advance or can vary without advance notice of

		the driver.
c2s[C-1:0]_data_stop_bcount[9:0]	Input	See c2s[C-1:0]_data_stop above.

Table 8-4 DMA Card to System Interface (Data Transfer)

PORT	TYPE	DESCRIPTION
c2s[C-1:0]_data_en	Output	Asserted to transfer one clock of data from a previously granted request. c2s_data_en may not be asserted continuously, so user logic should use c2s_data_en to provide each data word.
c2s[C-1:0]_data_first_req	Output	Asserted to mark the first c2s_data_en for each command data transfer. Only valid when c2s_data_en is asserted.
c2s[C-1:0]_data_last_req	Output	Asserted to mark the last c2s_data_en for each command data transfer. Only valid when c2s_data_en is asserted.
c2s[C-1:0]_data_first_desc	Output	Asserted to mark the first c2s_data_en for each Descriptor. Only valid when c2s_data_en is asserted.
c2s[C-1:0]_data_last_desc	Output	Asserted to mark the last c2s_data_en for each Descriptor. Only valid when c2s_data_en is asserted.
c2s[C-1:0]_data_first_chain	Output	Asserted to mark the first c2s_data_en of the first Descriptor in a chain/packet. Only valid when c2s_data_en is asserted. Unused and tied to zero for Packet DMA Engines.
c2s[C-1:0]_data_last_chain	Output	Asserted to mark the last c2s_data_en of the last Descriptor in a chain/packet. Only valid when c2s_data_en is asserted. When the DMA Direct Control interface is used for Block DMA, this output will only assert when a Descriptor finishes which had the "Last Descriptor in Chain" Descriptor bit set. Unused and tied to zero for Packet DMA Engines.
c2s[C-1:0]_data_remain[R-1:0]	Output	Number of ending bytes of c2s_data_data that are not needed and unused for the current c2s_data_en. Only non-zero on the final c2s_data_en in each request. R is the {4, 3, 2} when the core data width is {128, 64, 32} bits respectively.
c2s[C-1:0]_data_valid[R:0]	Output	Number of bytes that are needed and will be used for the current c2s_data_en. 2^R except for the final c2s_data_en in each request. Same, but opposite meaning of c2s_data_remain. R is the {4, 3, 2} when the core data width is {128, 64, 32} bits respectively.
c2s[C-1:0]_data_data[D-1:0]	Input	One word of data must be provided on c2s_data_data one clock following each c2s_data_en assertion. c2s_data_data width D is the same as the Core Data Width. c2s_data_data must be zero aligned such that the first requested data byte is located at byte 0. If user logic is implementing an addressed rather than a FIFO scheme user logic will in many cases need to read bytes from card memory for the first word which don't transfer on PCI Express because of a non-Core Data Width aligned c2s_data_addr starting address. Even if c2s_data_addr starts aligned, it can still get unaligned after the completion of an unaligned System Starting Address and Byte Count combination. The alignment problem is avoided if the system software can guarantee Core Data Width address alignment for all System Starting Address, Card Starting Addresses, and Byte

		Counts. This is unfortunately not practical for software in many cases.
c2s[C-1:0]_data_sop	Input	For Packet DMA only. Enables user hardware to mark the beginning of packets. Must be set (1) when c2s_data_data contains the first byte of a packet. Clear (0) otherwise. Same timing as c2s_data_data (one clock following the associated c2s_data_en).
c2s[C-1:0]_data_eop	Input	For Packet DMA only. Enables user hardware to mark the end of packets. Must be set (1) when c2s_data_data contains the last byte of a packet. Clear (0) otherwise. Same timing as c2s_data_data (one clock following the associated c2s_data_en).
c2s[C-1:0]_data_user_status[63:0]	Input	For Packet DMA only. Enables user hardware to optionally pass status information at the end of a packet from the user hardware design to user software. The value on c2s_data_user_status is captured when c2s_data_eop is asserted for a data transfer and is copied into the Descriptor's User Status field as part of the Descriptor completion status update. The DMA Engine does not use this field. Use of this field is optional. If unused, tie to 0.

Table 8-5 DMA System to Card Interface (Command)

PORT	TYPE	DESCRIPTION
s2c[S-1:0]_cmd_rst_n	Output	Active low DMA Engine reset. When 0 the DMA Engine is in reset and all user logic that interacts with the DMA Engine must be reset. Any outstanding DMA transactions will not complete.
s2c[S-1:0]_cmd_req	Output	Asserted by the DMA Engine to request permission to begin a new read request from system memory. When s2c_cmd_req is asserted, s2c_cmd_addr, s2c_cmd_bcount, and s2c_cmd_user_control indicate the Card Starting Address, Byte Count, and user control word for the request.
s2c[S-1:0]_cmd_ready	Input	The user asserts s2c_cmd_ready to signal acceptance of the request indicated on s2c_cmd_req, s2c_cmd_addr, s2c_cmd_bcount, and s2c_cmd_control. This indicates that the user has saved/no longer needs the information on s2c_cmd_addr, s2c_cmd_bcount, and s2c_cmd_control and the DMA Engine can make the associated read request.
s2c[S-1:0]_cmd_addr[63:0]	Output	Card Starting Address for the DMA operation. Only valid when s2c_cmd_req == 1. s2c_cmd_addr is a byte address. This value is copied directly from the DMA Descriptor for the first request of every Descriptor and is incremented by the previous transfer byte count with each subsequent request in the same Descriptor. s2c_cmd_addr is not necessary for FIFO applications so can be unused or re-tasked to provide control information such as which of several FIFOs data should transfer between. For Block DMA, 64-bits are available. For Packet DMA 36-bits are available.
s2c[S-1:0]_cmd_bcount[9:0]	Output	Number of bytes to transfer in the current DMA operation. Only valid when s2c_cmd_req == 1. This value is typically the max read request size in use (128, 256, or 512 bytes) but may be shorter at the beginning and end of a DMA Descriptor.
s2c[S-1:0]_cmd_user_control[31:0]	Output	Descriptor User Control (Packet DMA only). Only valid when s2c_cmd_req == 1. This value is copied directly from the User Control field of the DMA Descriptor and is the same for all requests that are part of the same Descriptor. Use of this field is optional and provides the user a mechanism to pass information between the user software and the hardware design.
s2c[S-1:0]_cmd_abort	Output	Asserted if a DMA operation is aborted by software request. All DMA transactions that are outstanding should be allowed to complete if possible. Once the user design has terminated and cleaned-up following the DMA abort, the user logic must assert s2c_cmd_abort_ack for one clock indicating that the abort is acknowledged and the user logic is ready for a new DMA operation or DMA reset to occur.
s2c[S-1:0]_cmd_abort_ack	Input	See s2c_cmd_abort description for details.
s2c[S-1:0]_cmd_stop	Input	Asserted instead of s2c_cmd_ready to request to stop the DMA chain/packet. When s2c_data_stop is asserted, s2c_cmd_stop_bcount[9:0] must contain the number of additional bytes that the user desires to transfer beyond those which were already promised via previous s2c_cmd_ready assertions. s2c_cmd_stop_bcount[9:0] must have a value

		between 0 (no additional transfers) and s2c_cmd_bcount[9:0]. If s2c_cmd_stop_bcount is non-zero, then an additional s2c_cmd_req will occur with s2c_cmd_bcount == s2c_cmd_stop_bcount. s2c_cmd_stop is used for abort conditions only and should not be used for normal operation.
s2c[S-1:0]_cmd_stop_bcount[9:0]	Input	See s2c_cmd_stop above.

Table 8-6 DMA System to Card Interface (Data Request/Transfer)

PORT	TYPE	DESCRIPTION
s2c[S-1:0]_data_req	Output	Asserted by the DMA Engine to request a data transfer. When s2c_data_req is asserted, s2c_data_addr and s2c_data_bcount indicate the Card Starting Address and Byte Count for the request.
s2c[S-1:0]_data_ready	Input	The user asserts s2c_data_ready to grant permission to the DMA Engine to execute the data transfer indicated on the s2c_data_req, s2c_data_addr, and s2c_data_bcount ports. Once a transaction has been granted, the data for the transaction will follow on the s2c_data_en and s2c_data_data ports when the DMA Engine can secure the data resources to complete the transfer. The time from a granted request to the first data will vary and no assumptions should be made.
s2c[S-1:0]_data_addr[63:0]	Output	Card Starting Address for the DMA operation. Only valid when s2c_data_req == 1. s2c_data_addr is a byte address. This value starts at the value contained in the DMA Descriptor and is incremented with each grant by the number of bytes which were granted. s2c_data_addr is not necessary for FIFO applications so can be unused or re-tasked to provide control information. For Block DMA, 64-bits are available. For Packet DMA 36-bits are available.
s2c[S-1:0]_data_bcount[9:0]	Output	Number of bytes to transfer in the current DMA operation. Only valid when s2c_data_req == 1. Requests are normally made at the maximum read request size which is typically 512 bytes, but the beginning and ending requests will vary according to the Descriptor System Starting Address offset and Byte Count.
s2c[S-1:0]_data_error	Output	Set (1) if an error occurred during the DMA transaction and the current data request data payload cannot be trusted to be correct. Clear (0) otherwise. Only valid when s2c_data_req == 1. s2c_data_error is asserted when a DMA read request was completed with one or more completions containing any of the following error conditions: • Completion Status != Successful • Completion marked Poisoned (EP == 1) • Completion for which the PCIe core detected and reported an ECRC error (only asserted for supported PCIe cores containing Advance Error Reporting Extended Capability and ECRC functionality)
s2c[S-1:0]_data_en	Output	Asserted to transfer one clock of data from a previously granted request. s2c_data_en may not be asserted continuously, so user logic should use s2c_data_en to provide each data word.
s2c[S-1:0]_data_first_req	Output	Asserted to mark the first s2c_data_en in each data request. Only valid when s2c_data_en is asserted.
s2c[S-1:0]_data_last_req	Output	Asserted to mark the last s2c_data_en in each data request. Only valid when s2c_data_en is asserted.
s2c[S-1:0]_data_first_desc	Output	Asserted to mark the first s2c_data_en in each Descriptor. Only valid when s2c_data_en is asserted.
s2c[S-1:0]_data_last_desc	Output	Asserted to mark the last s2c_data_en in each Descriptor. Only

		valid when s2c_data_en is asserted.
s2c[S-1:0]_data_first_chain	Output	Asserted to mark the first s2c_data_en in each DMA chain for Block DMA and to mark the start of packet (SOP) for Packet DMA. Only valid when s2c_data_en is asserted.
s2c[S-1:0]_data_last_chain	Output	Asserted to mark the last s2c_data_en in each DMA chain for Block DMA and to mark the end of packet (EOP) for Packet DMA. Only valid when s2c_data_en is asserted.
s2c[S-1:0]_data_remain[R-1:0]	Output	Number of ending bytes of s2c_data_data that are not valid for the current s2c_data_en. Only non-zero on the final s2c_data_en in each request. R is the {4, 3, 2} when the core data width is {128, 64, 32} bits respectively.
s2c[S-1:0]_data_valid[R:0]	Output	Number of ending bytes of s2c_data_data that are valid for the current s2c_data_en. 2^R except for the final s2c_data_en in each request. Same, but opposite meaning of s2c_data_remain. R is the {4, 3, 2} when the core data width is {128, 64, 32} bits respectively.
s2c[S-1:0]_data_data[D-1:0]	Output	One word of data is provided on s2c_data_data the same clock as the associated s2c_data_en assertion. Note that this data timing differs from the Card to System direction. s2c_data_data width D is the same as the Core Data Width. s2c_data_data is zero aligned such that the first valid data byte is located at byte 0. User logic implementing addressed resources will need to “shift-up” the data if the lower Card Address bits are non-zero. Even if s2c_data_addr starts aligned, it can still get unaligned after the completion of an unaligned System Starting Address and Byte Count combination. The alignment problem is avoided if the system software can guarantee Core Data Width address alignment for all System Starting Address, Card Starting Addresses, and Byte Counts. This is unfortunately not practical for software in many cases.
s2c[S-1:0]_data_user_control[63:0]	Output	For Packet DMA only. Enables user software to optionally pass control information at the start of a packet from the user software to user hardware design. The value on s2c_data_user_control is captured from the Descriptor User Control field and is output on s2c_data_user_control the same clock that s2c_data_first_chain (SOP) is output for the packet. The DMA Engine does not use this field. Use of this field is optional. If unused, tie to 0.

Table 8-7 DMA Card to System Interface Command Ports

PORT	TYPE	DESCRIPTION

c2s[C-1:0]_cfg_constants[63:0], s2c[S-1:0]_cfg_constants[63:0]	Input	DMA Engine Configuration Constants
		
		The Card to System and System to Card DMA Engines implement a high-degree of optional functionality including:
		• 32/64-bit System Address support
		• 32/64-bit Descriptor Address support (Block DMA Only)
		• 0 to 64-bit Card Address support (0-36-bit Packet DMA)
		
		DMA Engine size may be reduced on a per engine basis, by selectively disabling support for one or more of these features. Eliminating each of the features which can be controlled via this port results in significantly smaller DMA Engine size.
		
		The following fields are defined:
		• Bit[0] – Reserved. Tie to 0. Was Enable Sequence / Continue functionality in previous DMA Back-End versions
		• Bit[1] – Enable 64-bit System Address Support
		  ○ 1 == Enabled (32/64-bit)
		  ○ 0 == Disabled (32-bit only)
		• Bit[2] – Enable 64-bit Descriptor Pointer Support
		  ○ 1 == Enabled (32/64-bit; Block DMA Only)
		  ○ 0 == Disabled (32-bit only)
		• Bit[3] – Enable Block DMA Overlap
		  ○ 1 == Enabled (Block DMA Only); Up to 2 (instead of 1) Block DMA commands are overlapped resulting in higher DMA throughput; in this mode c2s_data_stop; s2c_cmd_stop may not be used and must be tied to 0.
		  ○ 0 == Disabled – Normal operation
		• Bits[7:4] – Reserved. Tie to 0.
		• Bits[14:8] – Card Address Size[6:0]
		  ○ Size of the DMA destination memory accessed by this DMA Engine
		  ○ Address size in bytes == 2^ Card Address Size (ex. Card Address Size == 28 == 256 Mbytes)
		  ○ FIFO and Packet applications normally set Card Address Size == 0
		  ○ For Block DMA values of 0 to 64 are valid
		  ○ For Packet DMA values of 0 to 36 are valid
		• Bit[15] – Reserved. Tie to 0.
		• Bits[21:16] – Implemented Byte Count Width[5:0]
		  ○ Maximum Descriptor byte size == 2^Implemented Byte Count Width-1
		  ○ For Block DMA values of 10 to 32 are valid
		  ○ For Packet DMA values of 10 to 20 are valid
		• Bits[31:22] – Reserved. Tie to 0.
		• Bits[38:32] – Implemented User Status Width[6:0]
		  ○ Number of bits to implement for optional User Status functionality. Set to 0 if unused
		• Bit[39] – Reserved. Tie to 0.
		• Bits[46:40] – Implemented User Control Width[6:0]
		  ○ Number of bits to implement for optional User Control functionality. Set to 0 if unused
		• Bit[47] – Disable_Descriptor_Updates

		○ 1 == Do not update system memory Descriptors with completion status when Descriptors complete (generally only useful when using the DMA Local Descriptor port) ○ 0 == Normal operation ○ Only relevant for Packet DMA; for Block DMA set to 0 • Bits[63:48] – Reserved. Tie to 0.

8.4 DMA Interface Example Transactions

Figure 8-3 illustrates an example Card to System DMA transaction.

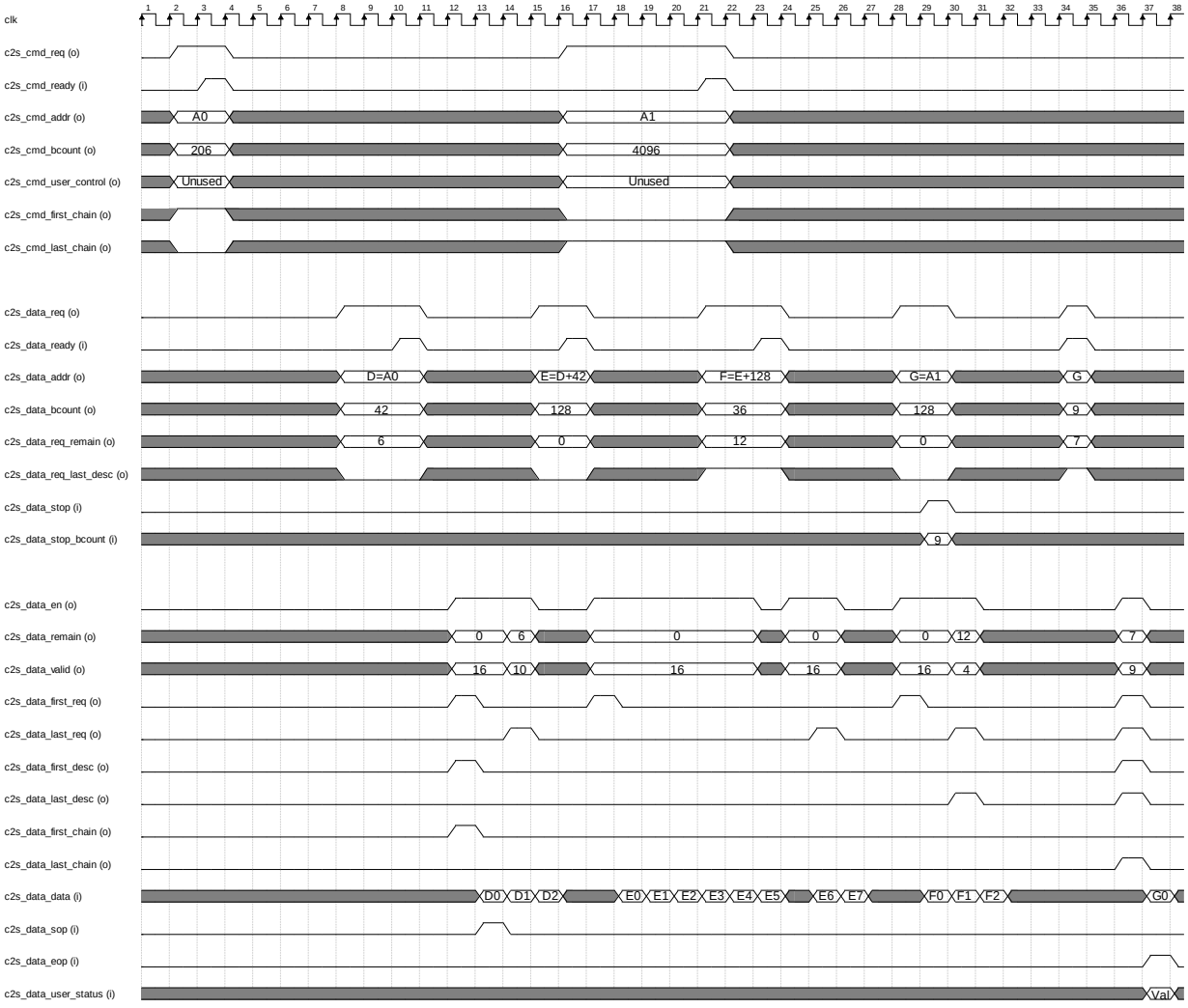


Figure 8-3 Card to System DMA Interface Example Transaction

The waveform in Figure 8-3 illustrates one packet spanning two Descriptors for a Card to System DMA Engine with Core Data Width == 128 bits (16 bytes). The second Descriptor is terminated early since it is assumed that only a portion of the second Descriptor was required to hold the user's packet. The behavior for other Core Data Widths is similar but less data transfers each clock.

The example waveform illustrates the affects of a System Starting Address and Byte Count which is not aligned to the PCI Express data payload size. PCI Express devices are prohibited from exceeding a maximum packet payload size (normally 128-512 bytes; often 128 bytes) and from crossing a 4 Kbyte system address boundary. To accommodate these requirements, the DMA Back-End Core will not create a packet for PCI Express that will cross a packet payload size system address boundary. This example assumes that the DMA Back-End Core is using a 128 byte payload size. A0 is assumed to have a 128 byte address offset of 86 bytes, and thus, the first request is for 42 bytes to reach a max payload size address boundary. The second (middle) request is for 128 bytes == max payload size and the final request in the Descriptor is the amount required to finish the Descriptor which is assumed to be 36 bytes.

The first Descriptor command request starts on cycle 2 with the assertion of c2s_cmd_req, c2s_cmd_addr, c2s_cmd_bcount, and c2s_cmd_user_control. c2s_cmd_first_chain is asserted to show the timing for Block DMA if this was the first Descriptor in a DMA Chain. c2s_cmd_first_chain is unused for Packet DMA. It is assumed that the user is unable to accept the request until cycle 3 when c2s_cmd_ready is asserted. Note c2s_cmd_user_control is unused in this DMA direction and is ignored.

Once the command has been accepted, the data requests begin on cycle 8 with the assertion of c2s_data_req, c2s_data_addr, c2s_data_bcount, c2s_data_req_remain, and c2s_data_req_last_desc. The user accepts the first data request on cycle 10 with the assertion of c2s_data_ready (c2s_data_ready could have been asserted as early as cycle 8). The data transfers for this request on cycles 13-15, one clock after the associated c2s_data_en on cycles 12-14. On cycle 12 c2s_data_first_req, c2s_data_first_desc, and c2s_data_first_chain (asserts for Block DMA only; unused for Packet DMA) assert. On cycle 14 c2s_data_last_req asserts indicating that this is last c2s_data_en for the current data request. On cycle 13 c2s_data_sop (Packet DMA only) is asserted by the user to mark this Descriptor as containing the start of a packet.

A second data request for the first Descriptor begins on cycle 15 followed by the associated c2s_data_en on cycles 17-22 and cycles 24-25. On cycle 23 the data flow is interrupted by c2s_data_en de-assertion to illustrate the behavior when the user is wait stated mid transfer.

The third and final data request for the first Descriptor begins on cycle 21 followed by the associated c2s_data_en on cycles 28-30. On cycle 30 c2s_data_last_desc asserts indicating that this is the last c2s_data_en for the current Descriptor.

A second Descriptor command request becomes pending on cycle 16 with the assertion of c2s_cmd_req. This Descriptor is a 4096 byte (typical OS page size) Descriptor. However, the user's packet requires only 9 bytes of this Descriptor, so the Descriptor is terminated early during the Descriptor's first c2s_data_req on cycle 28 when c2s_data_stop is asserted on cycle 29 with c2s_data_stop_bcount == 9. For Block DMA, this process would end the current DMA chain after transferring the final 9 bytes. For Packet DMA, this process causes the final Descriptor to be truncated to the size required to hold the packet (9 bytes). A final c2s_data_req for the Descriptor occurs on cycle 34 when c2s_data_bcount == the requested 9 bytes additional data transfer and c2s_data_ready is asserted. The data for this request transfers on cycle 37, one clock after the associated c2s_data_en on cycle 36. Also on cycle 37, c2s_data_eop is asserted to mark the end of the packet (Packet DMA only) and c2s_data_user_status is provided.

c2s_data_en should not be assumed to be contiguous for a data request as illustrated on cycle 23.

Figure 8-4 illustrates an example System to Card DMA transaction.

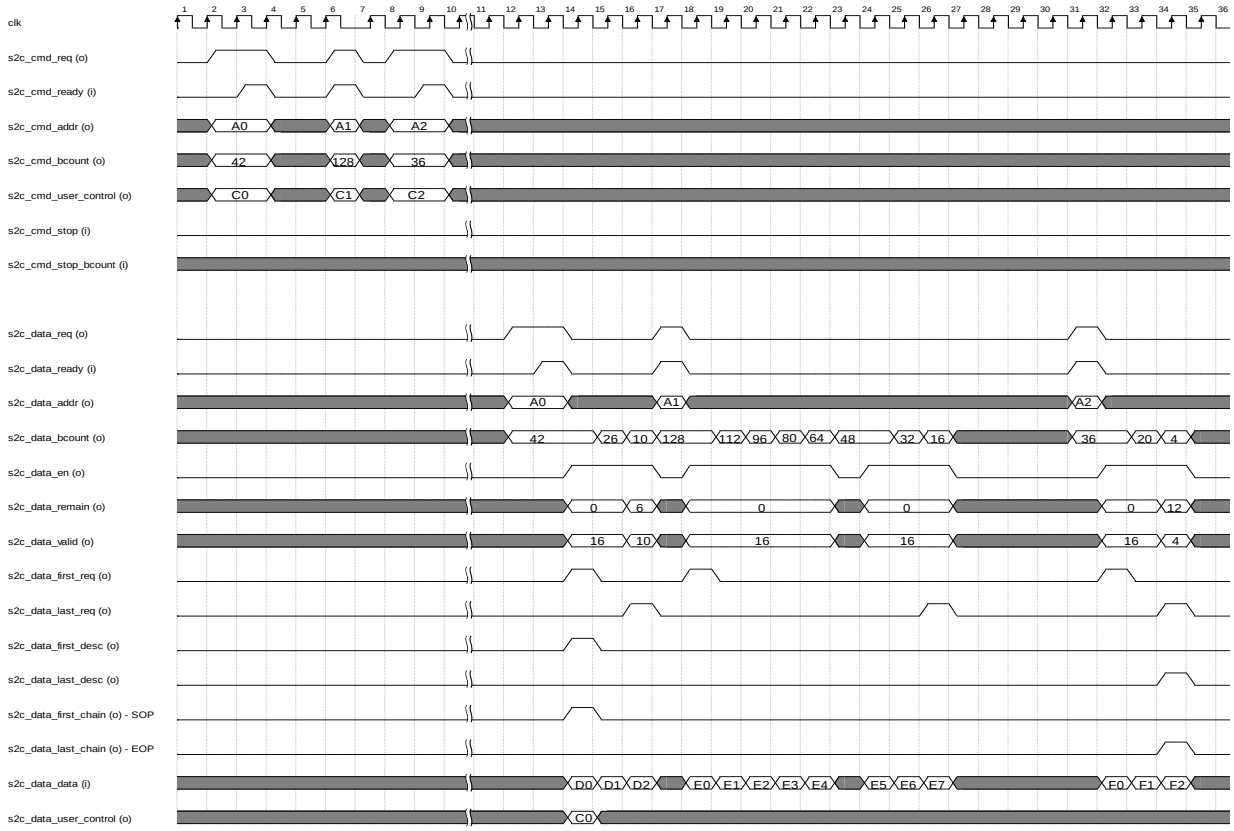


Figure 8-4 System to Card DMA Interface Example Transaction

The waveform in Figure 8-4 illustrates a small Descriptor execution for a System to Card DMA Engine with Core Data Width == 128 bits (16 bytes). The Descriptor contains both the start and end of a packet (Packet DMA). The behavior for other Core Data Widths is the same but less data transfers each clock.

The example waveform illustrates the affects of a System Starting Address and Byte Count which is not aligned to the PCI Express data payload size. PCI Express devices are prohibited from exceeding a maximum read request size (normally 512 bytes) and from crossing a 4 Kbyte system address boundary. To accommodate these requirements, the DMA Back-End Core will not read a packet from PCI Express that will cross a maximum read request size system address boundary. This example assumes that the DMA Back-End Core is using a 128 byte max read request size although the more common case is 512 bytes.

The first command request for the Descriptor starts on cycle 2 with the assertion of s2c_cmd_req, s2c_cmd_addr, s2c_cmd_bcount, and s2c_cmd_user_control. It is assumed that the user is unable to accept the request until cycle 3 when s2c_cmd_ready is asserted. Two additional commands are requested and are granted by the user on cycles 6 and 9 through the assertion of s2c_cmd_ready. In the System to Card DMA direction multiple simultaneous read requests are made to allow enough requested read data to be pending such that the read data latency from system memory can be swamped after the initial startup time. If the Descriptor byte count were longer, several initial s2c_cmd_req command phases would complete in rapid succession followed by single command requests as previous requests completed.

Once commands have been accepted, the corresponding read requests are made to PCI Express. PCI Express data reads normally have a pretty high degree of latency since they must obtain data from system memory which is normally in another chip in a distant PCI Express device. This is shown in the example as the DMA Interface being idle until cycle 12 when it is assumed that the first read data (data for the oldest read request) becomes available.

For PCI Express, due to PCI Express packet header and framing (5-7 DWORDs) being stripped before providing data to the user, at least 1 clock can go unused between packets at all Core Data Widths. This clock is used to advantage by the System to Card DMA interface between transfers such that the user does not have to handle overlapping data requests.

Large read requests are often completed using a series of 64 byte completions. Each completion has 5-8 DWORDs of header and framing which do not appear on this bus. Larger read requests such as a 512 byte read request may be completed as eight 64-byte completions for a total overhead of 40-64 bytes. Thus is is common to see large gaps between successive groups of s2c_data_en.

Note that s2c_data_bcount decrements with each s2c_data_en.

Note that s2c_data_first_chain indicates the start of a DMA chain for Block DMA and SOP for Packet DMA.
Note that s2c_data_last_chain indicates the end of a DMA chain for Block DMA and EOP for Packet DMA.

Note that s2c_data_user_control is only valid for Packet DMA when s2c_data_first_chain == 1.

8.5 Common DMA Register Block

A common DMA Register block is provided to aggregate DMA interrupt control and status information and to communicate common system status information.

Table 8-8 Common DMA Register Block

Byte Address	Description
0x3 - 0x0	DMA_Common_Control_and _Status[31:0] – Bit[0] Read/Write; others Read Only Bit[0] – Global DMA Interrupt Enable; this bit globally enables/disables interrupts for all DMA Engines. For a DMA Engine to generate an interrupt its individual interrupt enable and Global DMA Interrupt Enable must both be set. • 1 == Globally Enable DMA Interrupts • 0 == Globally Disable DMA Interrupts Bit[1] – DMA Interrupt Active; this bit reflects the state of the DMA interrupt hardware output. • 1 == A DMA interrupt is currently active and is not masked by Global DMA Interrupt Enable. • 0 == No DMA interrupts are active or they are masked by Global DMA Interrupt Enable. Bit[2] – DMA Interrupt Pending; same as DMA Interrupt Active, but without considering Global DMA Interrupt Enable. This bit can be used by software to know when an interrupt is pending that is masked. This bit is not used for interrupt generation and is for status information only. Bit[3] – Interrupt Mode • 1 == The DMA Back-End Core is in MSI-X/MSI Interrupt Mode • 0 == The DMA Back-End Core is in Legacy Interrupt Mode Bit[4] – Read/Write: User_Interrupt_Enable (optional feature enabled on request only) • This bit enables/disables the generation of user interrupts generated through the “user_interrupt” port. • 1 == Enable interrupt generation • 0 == Disable interrupt generation Bit[5] – User_Interrupt_Active (write a 1 to clear; in the case of a simultaneous set and clear operation, the set takes precedence) (optional feature enabled on request only) • 1 == User Interrupt is active • 0 == User Interrupt is inactive Bit[6] – MSI-X/MSI Mode – Only valid when Bit[3] == 1 (MSI-X/MSI Interrupt Mode) • 1 == The DMA Back-End Core is in MSI-X Interrupt Mode • 0 == The DMA Back-End Core is in MSI Interrupt Mode • Only relevant for designs that support MSI-X interrupts Bits[7] - Reserved Bits[10:8] – Max Payload Size Used[2:0]; the maximum payload size being used by the DMA Back-End Core. This may be different than the system-programmed Max Payload Size. This setting indicates the typical payload size for DMA Write Requests (Card to System direction). • 000 == 128 bytes • 001 == 256 bytes • 010 == 512 bytes • Others reserved Bit[11] – Reserved Bits[14:12] – Max Read Request Size Used[2:0]; the maximum read request size being used by the DMA Back-End Core. This may be different than the system-programmed Max Read Request Size. This setting indicates the typical read request size for DMA Read Requests (System to Card direction). • 000 == 128 bytes

	• 001 == 256 bytes • 010 == 512 bytes • Others reserved Bit[15] – Reserved Bits[23:16] – S2C_Interrupt_Status[7:0]; copy of the first 8 System to Card DMA Engine Interrupts Status registers. S2C_Interrupt_Status[i] corresponds to System to Card DMA Engine[i]; if System to Card DMA Engine[i] is not present then 0 is read. Provided to reduce the number of register reads required to determine the source of an interrupt. This register is READ ONLY. Interrupts are cleared by writing to the DMA Engine's Interrupt Status register. Bits[31:24] – C2S_Interrupt_Status[7:0]; copy of the first 8 Card to System DMA Engine Interrupts Status registers. C2S_Interrupt_Status[i] corresponds to Card to System DMA Engine[i]; if Card to System DMA Engine[i] is not present then 0 is read. Provided to reduce the number of register reads required to determine the source of an interrupt. This register is READ ONLY. Interrupts are cleared by writing to the DMA Engine's Interrupt Status register.
0x7 - 0x04	DMA Back-End Core Version[31:0] – DMA Back-End Core version. Incremented with feature additions and bug fixes.
0xB - 0x08	PCI Express Core Version[31:0] – Value of DMA Back-End Core port "mgmt_pcie_version". This field is used by the user to encode version information for the PCI Express Core that is connected to the DMA Back-End. Use of this field is optional. If unused, set mgmt_user_version = 0x00000000.
0xF - 0x0C	User Version[31:0] – Value of DMA Back-End Core port "mgmt_user_version". This field is used by the user to encode version information for the user's design. Use of this field is optional. If unused, set mgmt_user_version = 0x00000000.
0xFF - 0x10	Reserved.

8.6 Descriptor Engine

The Descriptor Engine is a central resource that accomplishes Descriptor reads for all Card to System and System to Card DMA register blocks.

The Descriptor Engine interleaves request using round robin arbitration. The Descriptor Engine is internal to the DMA Back-End Core and does not interact with user logic.

8.7 Descriptor Update Engine

The Descriptor Update Engine is a central resource that accomplishes Descriptor status update writes for all Packet DMA Engine register blocks.

The Descriptor Update Engine interleaves request using round robin arbitration. The Descriptor Update Engine is internal to the DMA Back-End Core and does not interact with user logic.

8.8 DMA Registers

Each System to Card and Card to System DMA Engine has an associated DMA Register module which software uses to control the DMA Engine. DMA Registers are different in Packet DMA and Block DMA engines.

8.8.1 Packet DMA Registers

The lower quarter of Base Address Register 0 (BAR0) is reserved for DMA Registers. During driver initialization, the driver reads the capabilities register for every possible DMA Engine location in BAR0 from 0x0 to 0x3FFF to determine how many engines are present and what their capabilities are. DMA Engine Register sets are located at offsets that are multiples of 0x100 (256) bytes. This is the same process described in the DMA Back-End User Guide.

The DMA Engine capabilities register indicates to software whether the engine is a Block DMA Engine, Packet DMA Engine, or not present. The programming model (DMA operation) for Block DMA and Packet DMA are different, so the driver must associate Block DMA Engines with the Block DMA API and Packet DMA Engines with the Packet DMA API.

When the DMA Back-End is licensed, the DMA Engines must be selected to be either Packet DMA or Block DMA Engines.

The Packet DMA Registers are described in Table 8-10. One such registers set is present for each Packet DMA Engine.

Table 8-9 Packet DMA Registers

Byte Address Offset	Description

0x3 - 0x0	DMA_Engine_Capabilities[31:0] – Read Only
	
	Identifies the capabilities of the associated DMA Engine.
	
	Bit[0] – Present
	• Indicates whether a DMA Engine register set is present at this location.
	• 1 == DMA Engine register set is present
	• 0 == DMA Engine register set is not present; the remaining portion of this register set is reserved
	Bit[1] – Engine Direction
	• Indicates the type of DMA Engine that this register set controls.
	  • 1 == Card to System DMA Engine; data flow is from card memory to system memory; the DMA Engine completes data transfers by transmitting bus write requests; system software normally thinks of this direction as the "read" direction since the data flow is the same as a CPU read
	  • 0 = System to Card DMA Engine; data flow is from system memory to card memory; the DMA Engine completes data transfers by transmitting bus read requests and receiving read completions; system software normally thinks of this direction as the "write" direction since the data flow is the same as a CPU write
	Bit[3:2] – Reserved
	Bit[7:4] – Engine Type
	• The remaining registers in this register set must be interpreted differently based upon the Engine Type:
	• == 0000 Block DMA Engine following Block DMA Register definition
	• != 0000 Packet DMA Engine following Packet DMA Register definition; individual bits are set if the corresponding feature is supported (must support at least one feature):
	  ○ xxx1 == Packet Send/Receive DMA API supported
	  ○ xx1x == Addressable Packet DMA API supported
	Bits[15:8] – Engine Number[7:0]
	• Identifies the unique DMA Engine Number of the Card to System or System to Card DMA Engine that this register set controls.
	Bits[22:16] – Card Address Size[6:0]
	• Identifies the size of the Card Address space that this DMA Engine transfers data to/from. This value is used by the driver and application to know how large an address space is accessible though this DMA Engine. The Card address space is (2^Card_Address_Size) bytes. This value is copied from the associated c2s/s2c_cfg_constants port field. Streaming (non-addressed) DMA applications should set the Card Address Size via c2s/s2c_cfg_constants to 0.
	Bit[23] – Reserved.
	Bits[29:24] – Descriptor Max Byte Count[5:0]
	• Identifies the maximum byte count that can be programmed into a Descriptor. If a system memory fragment size exceeds Descriptor Max Byte Count, then that fragment must be broken up into multiple Descriptors. The maximum byte count supported by a Descriptor is ((2^ Descriptor Max Byte Count Size)-1) bytes. However, it is recommended for best performance, to break Descriptors on binary multiple boundaries and to consider the maximum byte count per Descriptors to be (2^(Descriptor Max Byte Count Size-1)) bytes instead. Descriptor Max Byte Count is copied from the associated c2s/s2c_cfg_constants port field and is typically 20 for Packet DMA and 32 for Block DMA. 0 is a special case and indicates Descriptor Max Byte Count == 32.
	Bits[31:30] – Reserved.

Table 8-10 Packet DMA Registers (Continued)

Byte Address Offset	Description
0x7 -0x4	DMA Engine Control[31:0] – Bits[12:10] and reserved bits are Read Only; others are Read/Write Bit[0] – Interrupt_Enable • This bit enables/disables the generation of interrupts for this DMA Engine. Interrupts are only generated when Interrupt_Enable == 1 and one or more interrupt events have occurred. • 1 == Enable interrupt generation • 0 == Disable interrupt generation • If legacy interrupts (level-based) are in use, then the hardware interrupt line is asserted whenever Interrupt_Active == 1 and Interrupt_Enable == 1. • If MSI interrupts (edge-based) are in use, then an MSI interrupt message is transmitted whenever an interrupt event occurs and Interrupt_Enable == 1 on the clock that the interrupt event occurred. Bit[1] – Interrupt_Active (write a 1 to clear) • Interrupt_Active is set whenever an interrupt event occurs. Interrupt_Active is asserted independent to whether Interrupt_Enable is asserted. • 1 == Interrupt event occurred • 0 == Interrupt event did not occur • Interrupt_Active is set when a Descriptor completes which had the IRQ_On_Completion Descriptor control bit set (normal completion interrupt) or when a Descriptor Alignment Error, Descriptor Fetch Error, or SW_Abort_Error occurs (error interrupt). Bit[2] – Descriptor_Complete (write a 1 to clear) • 1 == Interrupt_Active was asserted because a Descriptor fully completed (including data transfer) which had the IRQ_On_Completion Descriptor control bit asserted (normal DMA completion interrupt). • 0 == Interrupt_Active was not asserted for this reason Bit[3] – Descriptor_Alignment_Error (write a 1 to clear) • 1 == Interrupt_Active was asserted because the DMA Engine was going to fetch the Descriptor at Reg_Next_Desc_Ptr for execution, but Reg_Next_Desc_Ptr was not aligned to start on a Descriptor-size address boundary. The DMA was aborted (DMA_Enable was forced low) without fetching or processing the Descriptor at Reg_Next_Desc_Ptr. • 0 == Interrupt_Active was not asserted for this reason Bit[4] – Descriptor_Fetch_Error (write a 1 to clear) • 1 == Interrupt_Active was asserted because the DMA Engine received an error during a Descriptor fetch (Descriptor read request returned completion status != successful) of the Descriptor at Reg_Next_Desc_Ptr. The DMA was aborted (DMA_Enable was forced low) without processing the Descriptor at Reg_Next_Desc_Ptr. • 0 == Interrupt_Active was not asserted for this reason Bit[5] – SW_Abort_Error (write a 1 to clear) • 1 == A DMA operation was aborted by software before it completed; this error occurs when the DMA Engine halts due to software clearing DMA_Enable prior to the DMA operation completing. • 0 == Interrupt_Active was not asserted for this reason Bit[6] - Reserved Bit[7] – Descriptor_Chain_End (write a 1 to clear) • 1 == The DMA Engine was going to fetch the Descriptor at Reg_Next_Desc_Ptr for execution, but Reg_Next_Desc_Ptr was 0 (indicating the end of the DMA chain was reached). The DMA was ended (DMA_Enable was forced low) because the end of the chain was reached.

	• 0 == DMA chain end condition has not occurred. • If software implements a finite chain of Descriptors, then the final Descriptor in the chain has DescNextDescPtr == 0. Descriptor_Chain_End indicates that all Descriptors in the chain were started and does not indicate that all DMA Descriptors in the chain fully completed. DMA_Running is provided for this purpose. • Descriptor_Chain_End is informational and does not affect Interrupt Active. To generate an interrupt when the final Descriptor in the DMA chain is completed, set the IRQ_On_Completion Descriptor control bit for the final Descriptor. Bit[8] – DMA_Enable • 1 == DMA Enabled ○ When DMA_Enable == 1, the DMA Engine monitors the Reg_Next_Desc_Ptr and Reg_SW_Desc_Ptr registers. When (Reg_Next_Desc_Ptr != Reg_SW_Desc_Ptr) and (Reg_Next_Desc_Ptr != 0), the DMA Engine fetches the Descriptor at Reg_Next_Desc_Ptr from system memory, passes the Descriptor to the DMA Engine for execution, and updates Reg_Next_Desc_Ptr with the value of the DescNextDescPtr (pointer to the next Descriptor) contained in the fetched Descriptor. The process repeats until Reg_Next_Desc_Ptr == Reg_SW_Desc_Ptr (wait condition) or Reg_Next_Desc_Ptr == 0 (DMA chain end condition). • 0 == DMA Disabled (power-on and reset default state) ○ DMA_Enable is cleared to disable the DMA Engine ○ DMA_Enable may be transitioned from 1 to 0 while a DMA operation is in progress to suspend/abort a DMA operation that has already been given to the DMA Engine to execute (by the driver advancing Reg_SW_Desc_Ptr); if this occurs all already started Descriptors will complete normally, but new Descriptors will not be started; transitioning DMA_Enable from 1 to 0 during DMA operation is not intended to be used as part of the standard DMA protocol and is provided for cases where the DMA Engine must be unexpectedly halted (system powering down; serious error detected, etc.); software should use control of Reg_SW_Desc_Ptr as the normal mechanism to pause the DMA operation. • DMA_Enable may not be transitioned from 0 to 1 while a DMA is already in process as indicated by DMA_Running == 1. Bit[9] – Reserved. Bit[10] – DMA_Running • 1 == DMA Engine is processing. • 0 == DMA Engine is Idle. • DMA_Running is provided for software to know when the DMA Engine is busy processing. DMA_Running is asserted when DMA_Enable == 1 and stays asserted until DMA_Enable == 0 and all DMA operations that were started while DMA_Enable was == 1 are completed. • When DMA_Enable is changed from 1 to 0 during a DMA operation, software must wait until DMA_Running == 0 before re-asserting DMA_Enable to start a new DMA operation or taking an action which will keep the pending DMA operation from completing (allowing the system to power down or enter a low power state for example) • DMA_Running is asserted both when the DMA Engine is executing Descriptors and when the DMA Engine is waiting for Reg_SW_Desc_Ptr != Reg_Next_Desc_Ptr. Bit[11] – DMA_Waiting • 1 == The DMA Engine has started execution of all of the Descriptors it has been given (started all Descriptors up to but not including Reg_SW_Desc_Ptr) and is waiting for more Descriptors to be made available by software. When DMA_Waiting == 1, the DMA Engine wants to fetch another Descriptor, but

cannot because Reg_Next_Desc_Ptr == Reg_SW_Desc_Ptr.
- 0 == DMA Engine is not waiting.
- DMA_Waiting is a real-time status signal and indicates the current status.

Bit[12] – DMA_Waiting_Persist – (Write a 1 to clear)
- Same as DMA_Waiting, but sets anytime DMA_Waiting == 1 and stays set until cleared by writing a 1.
- DMA_Waiting_Persist can be checked at the end of a DMA operation to determine whether the DMA Engine had to wait for software at some point in the DMA operation. DMA_Waiting_Persist == 1 indicates that the DMA Engine cannot achieve maximum throughput performance because it is being held back by the availability of Descriptors (software is processing Descriptors slower than the DMA Engine).

Bit[13] – Reserved

Bit[14] – DMA_Reset_Request
- Write a 1 to request that the user logic connected to the DMA Engine abort all outstanding operations, including allowing any outstanding DMA operations to complete either with valid or invalid data, in preparation for being reset. A write to this register causes the corresponding Streaming FIFO interface c2s[C-1:0]_abort/ s2c[C-1:0]_abort port to assert. After receiving a c2s[C-1:0]_abort_ack/ s2c[C-1:0]_abort_ack response on the streaming interface, DMA_Reset_Request is cleared indicating to software that the user design is ready to allow a reset to occur.

Bit[15] – DMA_Reset
- Write 1 to reset the DMA Engine and user logic connected to the DMA engine. The corresponding DMA Engine and DMA registers are reset and the corresponding c2s_cmd_rst_n/s2c_cmd_rst_n port is asserted for 16 clocks and then released in order to reset user logic.
- This bit is self clearing after the 16 clock reset pulse has been generated.

Bits[19:16] – Reserved

Bits[22:20] – Descriptor_Fetch_Error_Subclass – Additional information on the reason for a Descriptor_Fetch_Error assertion. One or more of these bits will be set when Descriptor_Fetch_Error is set:

Bit[20] – Unsuccessful Completion
- 1 == One or more DMA Descriptor fetch read data completions returned Completion Status != Successful or had a Completion Timeout Error
- 0 == No errors of this type

Bit[21] – Poisoned Completion
- 1 == One or more DMA Descriptor fetch read data completions were marked as Poisoned (EP ==1)
- 0 == No errors of this type

Bit[22] – ECRC Error Completion
- 1 == One or more DMA Descriptor fetch read data completions contained an ECRC error.
- 0 == No errors of this type

Bits[31:23] – Reserved

Table 8-10 Packet DMA Registers (Continued)

Byte Address Offset	Description
0xB -0x8	{Reg_Next_Desc_Ptr[31:5], 00000} – Bits[31:5] Read/Write when DMA_Running == 0; Read Only otherwise; resets to 0x0 32-bit system address pointer to the location of the next Descriptor to execute. Reg_Next_Desc_Ptr[31:5] is a 32-byte (Descriptor size) aligned address to guarantee that the entire Descriptor can be fetched in one read operation. Reg_Next_Desc_Ptr may be written by software only when DMA_Enable == 0 and DMA_Running == 0 indicating that the DMA Engine is idle. When the DMA Engine is not idle, the DMA Engine updates Reg_Next_Desc_Ptr with the value of the DescNextDescPtr field for each fetched Descriptor. Software should write Reg_Next_Desc_Ptr only to initialize the start of a new DMA Chain from the DMA Idle condition. Reg_Next_Desc_Ptr is used by the DMA Engine to keep track of the next Descriptor to start and can be used to indicate that prior Descriptors in the chain have been started. Reg_Next_Desc_Ptr does not indicate that prior Descriptors completed. Software should use the Reg_Completed_Desc_Ptr register to monitor which Descriptors have completed. Please see the DMA_Enable register bit, Reg_SW_Desc_Ptr, and Reg_Completed_Desc_Ptr description for additional detail.
0xF - 0xC	{Reg_SW_Desc_Ptr[31:5], 00000} – Read/Write - Bits[31:5] Read/Write; resets to 0x0 32-bit system address pointer to the location of the first Descriptor in the chain that is still “owned” by software. Descriptors in the chain between, but not including, Reg_Completed_Desc_Ptr and Reg_SW_Desc_Ptr are “owned” by the DMA Engine. Descriptors in the chain starting at Reg_SW_Desc_Ptr and following and Descriptors prior to and including Reg_Completed_Desc_Ptr are “owned” by software. The DMA Engine executes Descriptors in the DMA Chain up to, but not including, the Descriptor located at Reg_SW_Desc_Ptr and will pause execution if Reg_Next_Desc_Ptr == Reg_SW_Desc_Ptr until this condition is no longer true. Software should always keep at least one Descriptor under its ownership so that software can append to the DMA Chain if desired. Software is only allowed to modify DMA Descriptors for which software is the “owner”. The DMA Engine is only allowed to fetch, execute, and write DMA completion status to Descriptors for which the DMA Engine is the “owner”. Reg_SW_Desc_Ptr[31:5] is a 32-byte (Descriptor size) aligned address. Reg_SW_Desc_Ptr may be written by software at will, but the write mechanism must be guaranteed to update the entire 32-bit value in one write. Operating systems and host hardware normally guarantee atomic operations of DWORD size, but the Driver and Application must still ensure that only a single DWORD access is used to access this register. Please see the DMA_Enable register bit, Reg_Next_Desc_Ptr, and Reg_Completed_Desc_Ptr description for additional detail.

Table 8-10 Packet DMA Registers (Continued)

Byte Address Offset	Description
0x13 -0x10	{Reg_Completed_Desc_Ptr[31:5], 00000} – Bits[31:5] Read/Write when DMA_Running == 0; Read Only otherwise; resets to 0x0 32-bit system address pointer to the location of the last Descriptor that was completed by the DMA Engine. Reg_Completed_Desc_Ptr[31:5] is a 32-byte (Descriptor size) aligned address. Reg_Completed_Desc_Ptr may be written by software only when DMA_Running == 0 indicating that the DMA Engine is idle. When the DMA Engine is not idle, the DMA Engine updates Reg_Completed_Desc_Ptr when a Descriptor is completed and the Descriptor completion status has been written into the Descriptor. Software should set Reg_Completed_Desc_Ptr = 0 only to initialize the start of a new DMA Chain from the DMA Idle condition. Software reads of Reg_Completed_Desc_Ptr must be guaranteed to retrieve the entire contents of Reg_Completed_Desc_Ptr in one single read operation since hardware autonomously updates Reg_Completed_Desc_Ptr independent of software reads. Operating systems and host hardware normally guarantee atomic operations of DWORD size, but the Driver and Application must still ensure that only a single DWORD access is used to access this register. Please see the DMA_Enable register bit, Reg_SW_Desc_Ptr, and Reg_Next_Desc_Ptr description for additional detail.
0x17 - 0x14	{DMA_Active_Time[31:2], Sample_Ctr[1:0]} – Read Only DMA_Active_Time records the number of nanoseconds in the previous 1 second period that the DMA Engine was running (DMA_Running == 1). DMA_Active_Time includes the time required to fetch the first Descriptor, to fill the DMA data pipeline, any time waiting for software, system, or user hardware resources to be available, and the time required to finish all DMA data transfers on the DMA Engine interfaces. DMA_Active_Time[31:2] has a resolution of 4nS. Appending two zeros to the lower bits or multiplying by 4 yields the value in nanoseconds: For example, DMA_Active_Time[31:2] == 0000_0000_0000_0000_0000_0000_0001_01 == 5 * 4nS == 20nS or 0000_0000_0000_0000_0000_0000_0001_0100 == 20 nS. See DMA_Completed_Byte_Count for additional detail.

Table 8-10 Packet DMA Registers (Continued)

Byte Address Offset	Description
0x1B - 0x18	{DMA_Wait_Time[31:2], Sample_Ctr[1:0]} – Read Only DMA_Wait_Time records the number of nanoseconds in the previous 1 second period that the DMA Engine was running but the DMA Engine could not fetch a Descriptor because there were no more Descriptors that were made available (Reg_Next_Desc_Ptr == Reg_SW_Desc_Ptr). If DMA_Wait_Time == 0, then software DMA processing kept up or exceeded the hardware DMA throughput. If DMA_Wait_Time != 0, then software DMA processing limited hardware DMA throughput. DMA_Wait_Time[31:2] has a resolution of 4nS. Appending two zeros to the lower bits or multiplying by 4 yields the value in nanoseconds: For example, DMA_Wait_Time[31:2] == 0000_0000_0000_0000_0000_0000_0001_01 == 5 * 4nS == 20nS or 0000_0000_0000_0000_0000_0000_0001_0100 == 20 nS. See DMA_Completed_Byte_Count for additional detail.
0x1F - 0x1C	{DMA_Completed_Byte_Count[31:2], Sample_Ctr[1:0]} – Read Only DMA_Completed_Byte_Count records the number of bytes that transferred in the previous 1 second period that the DMA Engine was enabled. DMA_Completed_Byte_Count has a resolution of 4 bytes, so a value of DMA_Completed_Byte_Count [31:2] == 0000_0000_0000_0000_0000_0000_0001_01 == 5 * 4 bytes== 20 bytes or 0000_0000_0000_0000_0000_0000_0001_0100 == 20 bytes. NOTE: DMA operations with DMA length of less than 4 bytes will not show any data completed in this register as DMA_Completed_Byte_Count is truncated to a resolution of 4 bytes. DMA_Active_Time, DMA_Wait_Time, and DMA_Completed_Byte_Count are DMA performance registers that are intended to be read by software to determine the throughput of the DMA Engine, and to determine if software DMA processing is keeping up with the hardware DMA throughput. Each of these registers is updated once per second while the DMA Engine is enabled. Since software registers reads are asynchronous to the sample period, the same Sample_Ctr[1:0] field is provided for each of the DMA_Active_Time, DMA_Wait_Time, and DMA_Completed_Byte_Count registers to help software associate registers reads from the 3 registers with the values taken from the same sample period. Every time that a sample is taken, the current value of Sample_Ctr[1:0] is saved in DMA_Active_Time, DMA_Wait_Time, and DMA_Completed_Byte_Count with the corresponding sample data and then Sample_Ctr[1:0] is incremented by 1 (3 rolls over to 0). If software reads the three registers and all three reads have the same Sample_Ctr[1:0] value, then the sample data is all from the same sample period.
0x23 - 0x20	{30'h0, DMA_Interrupt_Control[1:0]} – Read/Write DMA_Interrupt_Control[1:0]: • 00 = Normal; interrupt when Descriptors complete that have Irq_On_Completion interrupt control bit set. This is the behavior exhibited by all previous versions of the DMA Back End.

	• 10 == Interrupt on EOP mode; interrupt when Descriptors complete that contain the end of a packet. This is a new option present in DMA Back End versions 01.05.08 (11/2/2009) and later. • 01, 11 == Reserved and are not allowed to be written. The driver and application generally only want to manage the DMA transaction in units of whole packets, so it is best for software performance to limit DMA completion interrupts to 1 interrupt per packet. For S2C Packet DMA, the DMA Driver knows the packet sizes in advance (application passes the driver the packets it wants transmitted) and creates the Descriptors for each packet individually. Normal (00) mode is typically used with the driver setting Irq_On_Completion only on the final Descriptor in a packet (the one containing the end of packet. Using Interrupt on EOP (10) mode results in the same result – one interrupt per packet - so either mode may be used. For C2S Packet DMA, the DMA Driver does not assume that the packet size is known in advance and implements a circular buffer to receive packets. Since Descriptors are used over and over, and it is unknown in advance where the end of received packets will be, it is necessary in Normal (00) mode to set Irq_On_Completion for every Descriptor. While this works just fine, CPU cycles are unnecessarily used to process interrupts for Descriptors that don't contain end of packet status. It is thus advantageous for C2S Packet DMA to use Interrupt on EOP (10) mode. This mode can dramatically reduce software overhead when packet size exceeds typical Descriptor size (4 KBytes).
0xFF - 0x20	Reserved.

8.8.2 Block DMA Registers

The register set for Card to System and System to Card DMA Engines is identical with the exception of the DMA Capability: Direction bit which identifies whether the DMA Registers control a Card to System DMA Engine or a System to Card DMA Engine.

DMA Registers are described in Table 8-10.

- Each DMA Engine has a 256 byte register space
- All registers are Read/Write unless otherwise specified
- Reserved bits return 0 when read

The DMA Register module interfaces to the system software via register writes and reads and to the Descriptor Engine to request that Descriptors be fetched from system memory. The DMA Register Descriptor fetch logic begins fetching the next Descriptor in the linked list as soon as the DMA Engine begins execution of the current Descriptor. In order for the Descriptor Engine to be able to fetch the next Descriptor before the current Descriptor completes, Descriptors should be > 2-4Kbytes to accommodate the usually substantial latency in reading from system memory. Typically operating systems fragment memory on >= 4Kbyte boundaries, so the Descriptor fetch operation does not limit the sustained throughput.

Table 8-10 DMA Engine Registers

Byte Address	Description
0x3 - 0x0	DMA_Engine_Capabilities[31:0] – Read Only Identifies the capabilities of the associated DMA Engine. Bit[0] – Present • Indicates whether a DMA Engine register set is present at this location. • 1 == DMA Engine register set is present • 0 == DMA Engine register set is not present; the remaining portion of this register set is reserved Bit[1] – Engine Type • Indicates the type of DMA Engine that this register set controls. • 1 == Card to System DMA Engine; data flow is from card memory to system memory; the DMA Engine will complete data transfers by transmitting write requests; system software normally thinks of this direction as the “read” direction since the data flow is the same as a CPU read • 0 = System to Card DMA Engine; data flow is from system memory to card memory; the DMA Engine will complete data transfers by transmitting read requests and receiving read completions; system software normally thinks of this direction as the “write” direction since the data flow is the same as a CPU write Bits[3:2] - Reserved Bit[7:4] – Engine Type • The remaining registers in this register set must be interpreted differently based upon the Engine Type: • == 0000 Block DMA Engine following Block DMA Register definition • != 0000 Packet DMA Engine following Packet DMA Register definition; individual bits are set if the corresponding feature is supported (must support at least one feature): ○ xxx1 == Packet Send/Receive DMA API supported ○ xx1x == Addressable Packet DMA API supported Bits[15:8] – Engine Number[7:0] • Identifies which Card to System or System to Card engine this register set controls. This register is defined to enable the possibility that multiple register sets may be instantiated to control the same DMA Engine. Software can tell that this is the case when two engines in the same direction have the same Engine Number. Bits[23:16] – Card Address Size[7:0] • Identifies the size of the Card Address space that this DMA Engine transfers data to/from. This value is used by the driver and application to know how large an address space is accessible though this DMA Engine. The Card address space is (2^Card_Address_Size) bytes. This value is copied from the associated c2s/s2c_cfg_constants port field. Bits[31:24] - Reserved

Table 8-10 Descriptor Engine Registers (Continued)

Byte Address	Description
0x7 -0x4	Control and Status[31:0] – Read/Write; bits[22:11] must be written to 1 to clear. Control interrupt generation and report DMA Engine status Bit[0] – Interrupt_Enable • This bit enables/disables the generation of interrupts for this DMA Engine. Interrupts are only generated on the completion of Descriptors which have Interrupt_Enable == 1 and for which one or more interrupt events have occurred. • 1 == Enable interrupt generation • 0 == Disable interrupt generation Bit[1] – Interrupt_Active (write a 1 to clear) • 1 == Interrupt is active • 0 == Interrupt is inactive Bits[7:2] - Reserved Bit[8] – Chain_Start • Setting this bit starts the Descriptor Engine. When this bit is set, the Descriptor Engine will begin fetching Descriptors from a linked-list of Descriptors starting at the Chain_Start_Descriptor_Pointer register address location. This bit will remain set until the Descriptor Engine starts the process of fetching the first Descriptor in the list. Once this bit is read == 0 after having been set to 1, the Chain_Start Descriptor_Pointer register can be changed to a different linked-list address and Chain Start can be set again so that the new linked-list of Descriptors will begin executing as soon as the current list completes. Bit[9] – DMA_Reset_Request (was Chain_Stop in previous DMA Back-End versions) • Software sets this bit to abort a DMA operation in process and to prepare hardware for a DMA reset. This bit remains set until the user hardware design acknowledges the request indicating that the user design is ready for another DMA operation or a DMA reset to occur. Bit[10] – Chain_Running • 1 == DMA Engine is processing a chain • 0 == Idle • Chain Running is set when a new Descriptor chain is started and remains set until the chain completes. Bit[11] – Chain_Complete • Set whenever a chain completes normally. This is the normal chain completion status and indicates that the entire chain completed without errors or being stopped early. Bit[12] – Chain_Error_Short • Set whenever a chain completes due to a Descriptor read completion error. See Chain_Error_Short_Subclass for additional information. Bit[13] – Chain_Software_Short • Set whenever a chain completes due to a software requested stop (Chain_Stop was asserted). Bit[14] – Chain_Hardware_Short • Set whenever a chain completes due to a hardware requested stop (DMA user-interface requested a stop) or a hardware error. See Chain_Hardware_Short Subclass for additional information. Bit[15] – Descriptor Alignment Error • Set whenever Chain Start is set but Chain_Start_Descriptor_Pointer is not aligned to a Descriptor width address boundary. The DMA Chain will not be started. Note that there is no error checking for Descriptors that are in the middle of the chain although they must also be aligned.

	Bits[19:16] – Chain_Hardware_Short Subclass – Additional information on the reason for a Chain Hardware Short Error. One or more of these bits will be set when Chain_Hardware_Short is set: Bit[16] – Unsuccessful Completion ○ 1 == One or more DMA read data completions returned Completion Status != Successful or had a Completion Timeout Error ○ 0 == No errors of this type Bit[17] – Poisoned Completion ○ 1 == One or more DMA read data completions were marked as Poisoned (EP ==1) ○ 0 == No errors of this type Bit[18] – ECRC Error Completion ○ 1 == One or more DMA read data completions contained an ECRC error. ○ 0 == No errors of this type Bit[19] – User Hardware Requested Stop ○ 1 == DMA stopped at request of user hardware (DMA user-interface requested a stop via data_stop/data_stop_bcount ports). Depending upon why the user design asserted data_stop, this may indicate a normal termination or error condition. ○ 0 == This No errors of this type Bits[22:20] – Chain_Error_Short_Subclass – Additional information on the reason for a Chain_Error_Short assertion. One or more of these bits will be set when Chain_Error_Short is set: Bit[20] – Unsuccessful Completion ○ 1 == One or more DMA Descriptor fetch read data completions returned Completion Status != Successful or had a Completion Timeout Error ○ 0 == No errors of this type Bit[21] – Poisoned Completion ○ 1 == One or more DMA DMA Descriptor fetch read data completion were marked as Poisoned (EP ==1) ○ 0 == No errors of this type Bit[22] – ECRC Error Completion ○ 1 == One or more DMA Descriptor fetch read data completions contained an ECRC error. ○ 0 == No errors of this type Bit[23] – DMA Reset • Write 1 to reset the DMA Engine and user logic connected to the DMA engine. The corresponding DMA Engine and DMA registers are reset and the corresponding c2s_cmd_rst_n/s2c_cmd_rst_n port is asserted for 16 clocks and then released in order to reset user logic. • This bit is self clearing after the 16 clock reset pulse has been generated. Bits[31:24] – Reserved

Table 8-10 Descriptor Engine Registers (Continued)

Byte Address	Description
0xF -0x8	Chain_Start_Descriptor_Pointer[63:0] – Read/Write Pointer to the first Descriptor in a linked-list of Descriptors. This is a 64-bit address in system memory. If only a 32-bit address is being used, then the upper 32-bits must be 0. This register must contain a valid pointer to a linked-list of Descriptors before Chain_Start can be set. Chain_Start_Descriptor_Pointer[4:0] must be 0 since Descriptors must be aligned on Descriptor size address boundaries.
0x13 - 0x10	Hardware_Time[31:0] – Read Only Total number of nanoseconds that was required for hardware to complete the last DMA chain. The Hardware_Time timer is started when the DMA Engine begins fetching the first Descriptor in a linked-list of Descriptors and is stopped when the final Descriptor in the link-list is executed or the chain is stopped early. The hardware throughput for a DMA operation can be calculated by dividing the total number of bytes moved in the DMA operation by the number of nanoseconds indicated by Hardware_Time.
0x17 - 0x14	Chain_Completed_Byte_Count[31:0] Number of bytes successfully transferred in the last complete chain. Updated and held on the completion of each DMA chain. If the chain stopped early, Chain_Completed_Byte_Count can be used to find out how much data transferred successfully.
0xFF -0x18	Reserved.

8.9 Aborting and Resetting DMA

8.9.1 Packet DMA

Software can abort an outstanding DMA operation before it is allowed to complete normally. To abort an outstanding DMA operation in as clean a manner as possible, the following process is followed:

- Software writes DMA_Enable = 0 and DMA_Reset_Request = 1
    - DMA_Enable == 0 causes the DMA Engine to stop queuing additional Descriptors. Any Descriptors which are already in process must still be allowed to complete. This is necessary because the DMA Engine typically has numerous transactions at various points in the system and user design and each of these transactions must complete in a controlled manner so that those data paths can be used for following operations. After writing DMA_Enable == 0, software reads DMA_Running until DMA_Running == 0 indicating that the DMA Engine completed all outstanding transactions and is idle.
    - Writing DMA_Reset_Request == 1 causes the user hardware logic connected to the DMA Engine to be notified of the software DMA abort request. User logic should allow the DMA operations on the DMA Interfaces to complete either with valid data, if available, or with invalid data when necessary. When the user logic is idle and ready for a reset to occur, user logic acknowledges the reset request and DMA_Reset_Request is cleared.
- After writing DMA_Enable = 0 and DMA_Reset_Request = 1, software polls until DMA_Running == 0 and DMA_Reset_Request == 0 indicating that the DMA Engine is idle and the user design connected to the DMA Engine is ready for a reset. Software should also implement a timeout mechanism where software will continue if DMA_Running or DMA_Reset_Request never become 0. The software timeout period should vary in accordance with how long it takes the user design to acknowledge an abort request and how large the DMA Descriptors in use are, but a typical value of 50mS should work for most applications.

Once (DMA_Running == 0 and DMA_Reset_Request == 0) or a timeout occurs, software writes DMA_Reset = 1 to reset the DMA Engine and user logic connected to the DMA Engine. The reset initializes the DMA Engine, and should initialize the user design, in the same manner as a power-on or system reset. DMA_Reset auto clears after the reset is complete. Note that DMA_Reset also resets the engine's DMA Registers, so any state information should be processed after DMA_Running ==0 and before DMA_Reset is asserted.

8.9.2 Block DMA

Software can abort an outstanding DMA operation before it is allowed to complete normally. To abort an outstanding DMA operation in as clean a manner as possible, the following process is followed:

- Software writes DMA_Reset_Request = 1
  - DMA_Reset_Request == 1 causes the DMA Engine to stop queuing additional Descriptors. Any Descriptors which are already in process must still be allowed to complete. This is necessary because the DMA Engine typically has numerous transactions at various points in the system and user design and each of these transactions must complete in a controlled manner so that those data paths can be used for following operations.
  - Writing DMA_Reset_Request == 1 causes the user hardware logic connected to the DMA Engine to be notified of the software DMA abort request. User logic should allow the DMA operations on the DMA Interfaces to complete either with valid data, if available, or with invalid data when necessary. When the user logic is idle and ready for a reset to occur, user logic acknowledges the reset request and DMA_Reset_Request is cleared. DMA_Reset_Request is also cleared once the DMA has successfully stopped (enough data was transferred to finish the outstaning DMA operations).
- After writing DMA_Reset_Request = 1, software polls until Chain_Running == 0 and DMA_Reset_Request == 0 indicating that the DMA Engine is idle and the user design connected to the DMA Engine is ready for a reset. Software should also implement a timeout mechanism where software will continue if Chain_Running or DMA_Reset_Request never become 0. The software timeout period should vary in accordance with how long it takes the user design to acknowledge an abort request and how large the DMA Descriptors in use are, but a typical value of 50mS should work for most applications.

Once (Chain_Running == 0 and DMA_Reset_Request == 0) or a timeout occurs, software writes DMA_Reset = 1 to reset the DMA Engine and user logic connected to the DMA Engine. The reset initializes the DMA Engine, and should initialize the user design, in the same manner as a power-on or system reset. DMA_Reset auto clears after the reset is complete. Note that DMA_Reset also resets the engine's DMA Registers, so any state information should be processed after DMA_Running ==0 and before DMA_Reset is asserted.

9 PCI Express Error Handling

9.1 Error Handling

The DMA Back End Core detects and handles the following types of PCI Express errors:

- Poisoned TLP Received – If the DMA Back-End receives a packet containing payload that is marked as poisoned (EP bit set in PCI Express Header) then the packet is handled as an error. Poisoned completions with data to Descriptor read requests cause the current DMA operation to be aborted. Poisoned completions with data to DMA read requests log error status, so software will know that the DMA data may not be valid. Poisoned target write requests will use the poisoned data. A Poisoned TLP Received error on a packet containing payload is signaled externally by asserting output port err_pkt_poison == 1 for 1 clock. The header of the Poisoned TLP is output on err_pkt_header the same clock that err_pkt_poison == 1 and should be connected to the front-end PCI Express Core's AER header logging logic (if present). err_pkt_poison should be connected to logic on the front-end PCI Express Core to report the error to the system. The recommended severity to report via the PCIe Core is Advisory Non-Fatal Error since the poison is handled in a manner that can allow software the opportunity to try to recover from the error. Poison only applies to payload data, so the poison (EP) bit is ignored and the packet is processed as if not poisoned if the packet does not contain payload.
- Unexpected Completion – The DMA Back-End Core receives incoming completions and matches them to the appropriate non-posted request based upon the completion's tag field. If a completion is received that does not match an outstanding request's tag, then this is an Unexpected Completion error and the completion is discarded. An Unexpected Completion error is also signaled externally by asserting output port err_cpl_to_closed_tag for 1 clock. err_cpl_to_closed_tag should be connected to logic on the front-end PCI Express Core to report the error to the system. The recommended severity to report is Advisory Non-Fatal Error (assume that the completion has been misrouted). The header of the Unexpected Completion is output on err_pkt_header the same clock that err_cpl_to_closed_tag == 1 and should be connected to the front-end PCI Express Core's AER header logging logic (if present).
- Completion Timeout – The DMA Back-End Core implements a completion timeout timer that fires approximately every 16mS. If a non-posted request does not receive a completion (has not made forward progress) between 2 timeout periods, then the DMA Back End assumes that the request will never complete and terminates the request with error status. The error status is passed to the DMA Engines and DWORD master for reporting to the user/software. A Completion Timeout error is also signaled externally by asserting output port err_cpl_timeout == 1 for 1 clock. err_cpl_timeout should be connected to logic on the front-end PCI Express Core to report the error to the system. The recommended severity to report via the PCIe Core is Non-Fatal Error since this is a serious system error.

9.2 Error Reporting Interface

Table 9-1 Error Reporting Interface

PORT	TYPE	DESCRIPTION
err_pkt_poison	Output	Asserted == 1 for 1 clock whenever a TLP with payload is received that has the Poison Indicator (EP) bit set.
err_cpl_to_closed_tag	Output	Asserted == 1 for 1 clock whenever a completion is received with a tag that does not match an outstanding request.
err_cpl_timeout	Output	Asserted == 1 for 1 clock whenever an outstanding non-posted request (read, I/O Write, Cfg Write) fails to be completed within the allotted timeout period.
err_pkt_header[127:0]	Output	The same clock that err_pkt_poison == 1 or err_cpl_to_closed_tag ==1, err_pkt_header contains the header of the TLP that contained the error. If both errors occur simultaneously, then the err_pkt_poison header takes priority and is reported.
cpl_tag_active	Output	cpl_tag_active == 1 whenever the DMA Back-End Core has outstanding non-posted requests (request have been made, but have not yet completed). This is not an error signal. The value of cpl_tag_active should be reflected in the front-end PCI Express Core's "Transactions Pending" configuration register bit in the Device Status Register (offset 0xA of PCI Express Capability Structure).

See Section 9.1 for additional detail on when these ports are used to indicate error conditions.

10 DMA Direct Control Interface

In addition to a Descriptor Engine, each System to Card and Card to System DMA Engine has a DMA Direct Control Interface which can be used by user logic to directly control the DMA Engine. The user uses the DMA Direct Control Interface to execute Descriptors serially and to receive DMA completion status.

Either the built-in DMA Register functionality or the DMA Direct Control Interface may be used to control the DMA Engine, but not both at the same time. Before it is permitted to switch DMA control between Registers and the DMA Direct Control Interface, all outstanding DMA transactions must be allowed to fully complete.

If the DMA Direct Control Interface is unused, then all input ports must be tied to 0 and all outputs should be ignored.

10.1 DMA Direct Control Interface Port Descriptions

The DMA Direct Control Interface ports are described in Table 10-1. The port name prefix for System to Card DMA Engines, is "s2c" while the port name prefix for Card to System DMA Engines is "c2s". There is one DMA Direct Control Interface for each DMA Engine. In the port names in Table 7-1, "S" is the number of implemented System toCard DMA Engines and "C" is the number of implemented Card to System DMA Engines.

Table 10-1 DMA Direct Control Interface

PORT	TYPE	DESCRIPTION
s2c[S-1:0]_desc_req/ c2s[C-1:0]_desc_req	Input	Asserted to request that a Descriptor be executed
s2c[S-1:0]_desc_ready/ c2s[C-1:0]_desc_ready	Output	Asserted for one clock cycle when the DMA Engine accepts the currently requested Descriptor. Each DMA Direct Control Interface transaction executes one Descriptor and then returns. s2c_desc_req/c2s_desc_req must be de-asserted when s2c_desc_ready/c2s_desc_ready is asserted unless a new Descriptor is desired to be executed as soon as the current one completes.
s2c[S-1:0]_desc_ptr[31:0]/ c2s[C-1:0]_desc_ptr[31:0]	Input	When s2c_desc_req/c2s_desc_req is asserted, s2c_desc_ptr/c2s_desc_ptr accepts a 32-bit value that will be passed unmodified onto the desc_done_status port when the Descriptor DMA completes. This field is used for DMA Register DMA control to store the system address where the Descriptor is located so that the Descriptor can be updated with DMA completion status when it completes. For the DMA Direct Control Interface, this field is unused but can be useful for user logic to associate Descriptor Requests with Descriptor Completions. This port is unused for Block DMA.
s2c[S-1:0]_desc_data[255:0]/ c2s[C-1:0]_desc_data[255:0]	Input	When s2c_desc_req/c2s_desc_req is asserted, s2c_desc_data/c2s_desc_data must contain the Descriptor that is desired to be executed. The Descriptor Format is described in Section 8.2.
s2c[S-1:0]_desc_done/ c2s[C-1:0]_desc_done	Output	Asserted for one clock when a previously accepted Descriptor has completed.
s2c[S-1:0]_desc_channel[7:0]/ c2s[C-1:0]_desc_done_channel[7:0]	Output	Reserved. Ignore.

s2c[S-1:0]_desc_done_status[63:0]/ c2s[C-1:0]_desc_done_status[63:0]	Output	When s2c_desc_done/c2s_desc_done, the status of the completed Descriptor is placed on s2c_desc_status / c2s_desc_status according to the following format by DMA type: Packet DMA • Bit[0] == 1 DMA Successful; 0 otherwise • Bit[1] == 1 DMA Completed short due to software request; 0 otherwise • Bit[2] == 1 DMA Completed short due to user hardware request; 0 otherwise • Bit[3] == 1 DMA completion should cause an interrupt to occur (IRQOnCompletion / Interrupt_on_Completion was set for this Descriptor) • Bit[4] == 1 An error was detected; 0 otherwise • Bit[5] – Reserved • Bit[6] == 1 if Descriptor contains the end of packet (EOP); only valid for Card to System DMA • Bit[7] == 1 if Descriptor contains the start of packet (SOP); only valid for Card to System DMA • Bit[8] == 1 if Descriptor UserStatus[31:0] == 0; only valid for Card to System DMA • Bit[9] == 1 if Descriptor UserStatus[63:32] == 0; only valid for Card to System DMA • Bit[10] == 1 if Descriptor UserStatus[31:0] should be updated in the Descriptor; 0 otherwise; only used for DMA Registers; only valid for Card to System DMA • Bit[11] == 1 if Descriptor UserStatus[63:32] should be updated in the Descriptor; 0 otherwise; only used for DMA Registers; only valid for Card to System DMA • Bits[31:12] - Reserved • Bits[51:32] == Descriptor Completed Byte Count[19:0]; number of bytes completed in this Descriptor • Bits[63:52] - Reserved • Bits[95:64] == c2s/s2c_desc_ptr value provided when the Descriptor was requested on this interface • Bits[159:96] == c2s_data_user_status[63:0] received from user hardware when completing this Descriptor; only valid for C2S Packet DMA containing EOP. Block DMA • Bit[0] == 1 DMA Successful; 0 otherwise • Bit[1] == 1 DMA Completed short due to software request; 0 otherwise • Bit[2] == 1 DMA Completed short due to user hardware request; 0 otherwise • Bit[3] == 1 An error was detected; 0 otherwise • Bit[31:4] - Reserved • Bits[63:32] == Descriptor Completed Byte Count[31:0]; number of bytes completed in this Descriptor • Bits[159:64] - Reserved

s2c[S-1:0]_desc_abort/ c2s[C-1:0]_desc_abort	Input	Asserted to request that previously granted Descriptor(s) be stopped early. The DMA operations will be stopped at the next possible checkpoint. All outstanding DMA operations must be allowed to complete. s2c/c2s_desc_abort must remain asserted until s2c/c2s_desc_abort_ack is asserted indicating that the user design has completed or flushed the outstanding DMA operations and is ready to continue with a new DMA operation or a DMA reset.
s2c[S-1:0]_desc_abort_ack/ c2s[C-1:0]_desc_abort_ack	Output	See s2c/c2s_desc_abort for details.
s2c[S-1:0]_desc_rst_n/ c2s[C-1:0]_desc_rst_n	Input	Active low DMA Engine reset. Asserting this input will cause the DMA Engine and user hardware to be reset. Before asserting this port, the DMA operations should first be stopped through s2c/c2s_desc_abort and s2c/c2s_desc_abort_ack.

!! When the DMA Direct Control interface is used with Packet DMA Engines, c2s/s2c_cfg_constants[47] == Disable_Descriptor_Updates should be set to 1 or Descriptor Update writes will be made to system memory to update the Descriptor with the completion status. This is generally not desired when the DMA Direct Control interface is being used since the Descriptors are provided locally rather than read from system memory.

10.2 DMA Direct Control Interface Example Transactions

Figure 10-1 illustrates an example DMA Direct Control Interface Example transaction for a System to Card DMA Engine DMA Direct Control Interface. The interface operation is identical for Card to System DMA Engine DMA Direct Control Interfaces.

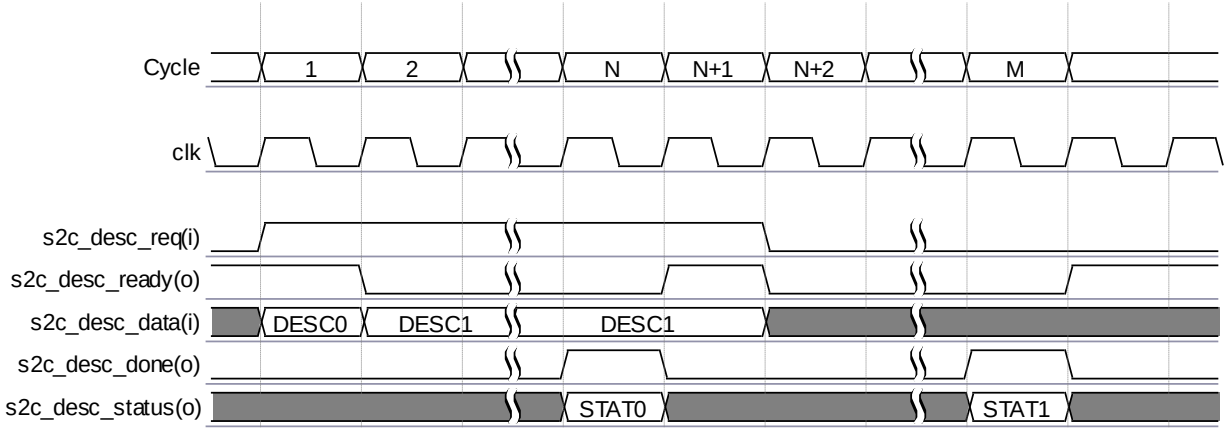


Figure 10-1 DMA Direct Control Interface Example Transaction

A DMA Direct Control Interface request begins on cycle 1 with the assertion of s2c_desc_req. On the same cycle the Descriptor (DESC0) is placed on s2c_desc_data. It is assumed that the associated DMA Engine is currently idle, so accepts the request the same cycle through the assertion of s2c_desc_ready on cycle 1.

It is assumed that the user wishes to start another DMA Direct Control operation as soon as the current one finishes, so s2c_desc_req is left asserted and s2c_desc_data is updated to the new Descriptor (DESC1) on cycle 2. The DESC0 DMA operation completes on Cycle N as indicated by s2c_desc_done == 1 and the completion status is indicated on s2c_desc_status on the same cycle.

On cycle N+1, the pending request for DESC1 is granted through the assertion of s2c_desc_ready == 1. It is assumed that an additional Descriptor operation is not desired immediately, and thus s2c_desc_req is de-asserted on cycle N+2.

The DESC1 DMA operation completes on Cycle M as indicated by s2c_desc_done == 1 and the completion status is indicated on s2c_desc_status on the same cycle.

11 Management Interface

11.1 Management Interface Port Descriptions

The DMA Back-End connects to several different PCI Express cores to implement the low-level bus functionality required to connect over PCI Express. This includes the Configuration Registers required for software to configure and control the device. The purpose of the Management Interface is to configure the DMA Back-End to operate in accordance with the configuration information programmed by software into the PCI Express device and to provide error and status information.

Management Port Descriptions (Miscellaneous):

- rst_n – input
  - Active low asynchronous reset to reset all DMA Back-End logic. rst_n should be connected to the PCI Express Core data_link_layer_up or equivalent signal.
- clk – input
  - Positive edge clock used to clock all interfaces of the DMA Back-End. The DMA Back-End clock must be the same clock as the PCI Express Core including any rate changes required as a result of shifting between PCI Express speeds of operation. If the clock rate changes during operation, mgmt_clk_period_in_ns must change accordingly with the clock period.
- mgmt_mst_en – input
  - (1) The Bus Master Enable Configuration register bit is set enabling master traffic to be generated. (0) Master traffic may not be generated.
- mgmt_msi_en – input
  - Indicates whether the PCI Express Core has been configured by the system to use MSI Interrupts (1) or Legacy Interrupts (0). MSI interrupts are event (edge) based while Legacy interrupts are level-based.
  - When mgmt_msi_en == 1 (MSI), mgmt_interrupt is strobed high for one clock cycle whenever an MSI interrupt write packet should be transmitted.
  - When mgmt_msi_en == 0 (Legacy), mgmt_interrupt is asserted and held high until the interrupt has been cleared by system software.
- mgmt_interrupt – output
  - Active high interrupt output to PCI Express Core to enact an interrupt. See mgmt_msi_en for additional information. DMA interrupts occur on this port.
- user_interrupt – input (optional feature enabled on request only)
  - Activ e high user interrupt input. See mgmt_msi_en for additional information.
- mgmt_max_payload_size[2:0] – input (PCI Express Only)
  - Value of the PCI Express Core Max Payload Size Configuration Register:
    - 000 – 128 bytes, 001 – 256 bytes, 010 – 512 bytes, 011 – 1024 bytes, 100 ,- 2048 bytes, 101 – 4096 bytes, 110 – Reserved, 111 – Reserved
    - The DMA Back-End issues DMA write requests using the system configured maximum payload size or a maximum of 512 bytes.
- mgmt_max_rd_req_size[2:0] – input (PCI Express Only)
  - Value of the PCI Express Core Max Read Request Size Configuration Register:
    - 000 – 128 bytes, 001 – 256 bytes, 010 – 512 bytes, 011 – 1024 bytes, 100 ,- 2048 bytes, 101 – 4096 bytes, 110 – Reserved, 111 – Reserved
    - The DMA Back-End issues DMA read requests using the system configured maximum read request size or a maximum of 512 bytes.
- mgmt_clk_period_in_ns[7:0]
  - Clock period of the DMA Back-End Core clock input in nanoseconds. Used for fixed-time calculations. Must match the clock period of the DMA Back-End Core at all times. If the PCI Express Core changes frequency of operation and this affects the DMA Back-End clock then this port must be updated with the changed clock period.
- mgmt_cfg_id[15:0] – input
  - Requestor/Completer ID which has been assigned to the PCI Express Core. This value is used as the Requestor/Completer ID for PCI Express packets which contains these fields. It is critical that this port be assigned the correct value or transactions will be misrouted causing system errors to occur.

The DMA Back-End issues DMA read requests and, in some cases, DWORD master read requests and receives the associated completions. PCI Express requires that read requests not be issued unless there is Receive Completion buffer space available to receive all of the associated completions. The following ports tell the DMA Back-End the characteristics of the PCI Express Core Receive Completion Buffer. It is critical that the values provided be accurate for the PCI Expresss Core in use. The DMA Back-End tracks outstanding read requests and completions to ensure that reads are not made if there are not completion resources to receive the resulting completions.

Management Port Descriptions (Completion Credits; PCI Express Only):

- mgmt_ch_credits[7:0] – Number of completion header credits implemented by the receive buffer of the PCI Express Core. See PCI Express Core User Guide for details. Note that PCI Express does not allow the advertisement of more than 127 Header Credits.
- mgmt_cd_credits[11:0] – Number of completion data payload credits implemented by the receive buffer of the PCI Express Core. See PCI Express Core User Guide for details. Note that PCI Express does not allow the advertisement of more than 2047 Data Credits.
- mgmt_ch_infinite – 0 == Check for required CH credits before issuing read requests (recommended). 1 == Assume that CH credits are infinite (mgmt_ch_credits not relevant in this case) and do not check CH credits. A value of 1 is not recommended unless it is known to be impossible to overflow the PCI Express Core's receive completion buffer and the design would otherwise be undesirably limited in performance due to a small receive completion buffer.
- mgmt_cd_infinite – 0 == Check for required CD credits before issuing read requests (recommended). 1 == Assume that CD credits are infinite (mgmt_cd_credits not relevant in this case) and do not check CD credits. A value of 1 is not recommended unless it is known to be impossible to overflow the PCI Express Core's receive completion buffer and the design would otherwise be undesirably limited in performance due to a small receive completion buffer.

Management Port Descriptions (Completion Timeout Configuration; PCI Express 2.0 Only):

- mgmt_adv_cpl_timeout_disable – output
  - Connect to PCI Express Core's PCI Device Capabilities2: bit[4] Configuration Register : Completion Timeout Disable Supported. 1 == Advertise support for disabling completion timeouts; 0 == advertise that completion timeout disabling is not supported. PCI Express 2.0 requires that this feature be supported, so this port is asserted (1).
- mgmt_adv_cpl_timeout_value[3:0] – output
  - Connect to PCI Express Core's PCI Device Capabilities2: bit[3:0] Configuration Register : Completion Timeout Ranges Supported. Indicates which timeout ranges are supported. The DMA Back-End only supports the default 50uS to 50 mS timeout range, so this port is 0x0.
- mgmt_cpl_timeout_disable – input
  - Connect to PCI Express Core's PCI Device Control2: bit[4] Configuration Register : Completion Timeout Disable. 1 == Disable completion timeouts; 0 == do not disable.
- mgmt_cpl_timeout_value[3:0] – input
  - Connect to PCI Express Core's PCI Device Control2: bit[3:0] Configuration Register : Completion Timeout Value. Indicates which timeout ranges is selected. The DMA Back-End only supports the default 50uS to 50 mS timeout range, so this port should always be 0x0.

Management Port Descriptions (Completion Timeouts):

- err_cpl_to_closed_tag – output
  - Strobed for one clock cycle whenever a completion was received to a tag which was closed. This is a reported error. Connect to PCI Express Core's error reporting ports to log the error.
- err_cpl_timeout – output
  - Strobed for one clock cycle whenever a request did not receive a completion within the allowed time. This is a reported error. Connect to PCI Express Core's error reporting ports to log the error.
- cpl_tag_active – output
  - Connect to PCI Express Core's Device Status : bit[6] Configuration Register : Transactions Pending. Asserted (1) whenever requests are outstanding that are waiting for completions.

Management Port Descriptions (Version)

- mgmt_version[31:0] – output
  - DMA Back-End Core Version
  - Incremented with feature additions and bug fixes
  - The value on this port is also in the Common Registers.
- mgmt_pcie_version[31:0] – input
  - Version of the PCI Express Core connected to the DMA Back-End. This port is optional. The value on this port is reflected in the Common Registers.
- mgmt_user_version[31:0] – input
  - Version of the User design connected to the DMA Back-End. This port is optional. The value on this port is reflected in the Common Registers.

12 Master Interface

The Master Interface enables the user to generate 0-1 DWORD payload requests to access remote system information in a simple manner. The Master Interface does not support bursts or overlapped transactions.

12.1 Master Interface Port Descriptions

The Master Interface ports are described in Table 12-1.

Table 12-1 Master Interface

Port	Type	Description
mst_req	Input	Asserted to request a master transaction be processed. mst_addr, mst_msg_code, mst_data, and mst_be must be valid when mst_req is asserted. Once a request has been made, it must remain until it is granted.
mst_ready	Output	Asserted when a mst_req request is granted. For requests that do not require completions (memory writes and message requests), mst_ready will be asserted as soon as the request is passed to the PCI Express Core. For requests which require completions, mst_ready is asserted when the completion is returned.
mst_type[6:0]	Input	Type of request (same mapping as Fmt, Type in PCI Express): • 00_00000 Memory Read (32-bit) • 01_00000 Memory Read (64-bit) • 10_00000 Memory Write (32-bit) • 11_00000 Memory Write (64-bit) • 00_00010 I/O Read • 10_00010 I/O Write • 00_00100 Configuration Read Type 0 (Root Complex Only) • 10_00100 Configuration Write Type 0 (Root Complex Only) • 00_00101 Configuration Read Type 1(Root Complex Only) • 10_00101 Configuration Write Type 1 (Root Complex Only) • 01_10rrr Message Request (rrr == routing) • 11_10rrr Message Request with Data (rrr == routing) • All others are reserved and cannot be used
mst_addr[63:0]	Input	PCI Express Address; required for Memory, I/O, and Configuration requests; not used for Message requests; if a 32-bit address is used, the upper 32-bits must be set to 0x0
mst_msg_code[7:0]	Input	PCI Express Message Code; required for Message requests; not used for Memory, I/O, or Configuration requests

Table 12-1 Master Interface (Continued)

Port	Type	Description
mst_data[31:0]	Input	Write request data payload; only used for write requests and Message Requests with Data
mst_be[3:0]	Input	Write/read request byte enables; required for all requests except message requests. Active high. 0xF == all bytes, 0x1 == byte0 only.
mst_done	Output	Asserted when a request completes. When mst_done == 1, mst_status and mst_rd_data contain the status and read data (for read requests) for requests requiring completions.
mst_status[2:0]	Output	For requests requiring completions (all reads and configuration write requests) contains the read completion status: • 000 == Succesful • 001 == Unsupported Request • 010 == Configuration Request Retry Status • 100 == Completer Abort • All others reserved
mst_rd_data[31:0]	Output	Read data returned for read requests

12.2 Master Interface Example Transactions

Figure 0-1illustrates Master Interface example transactions.

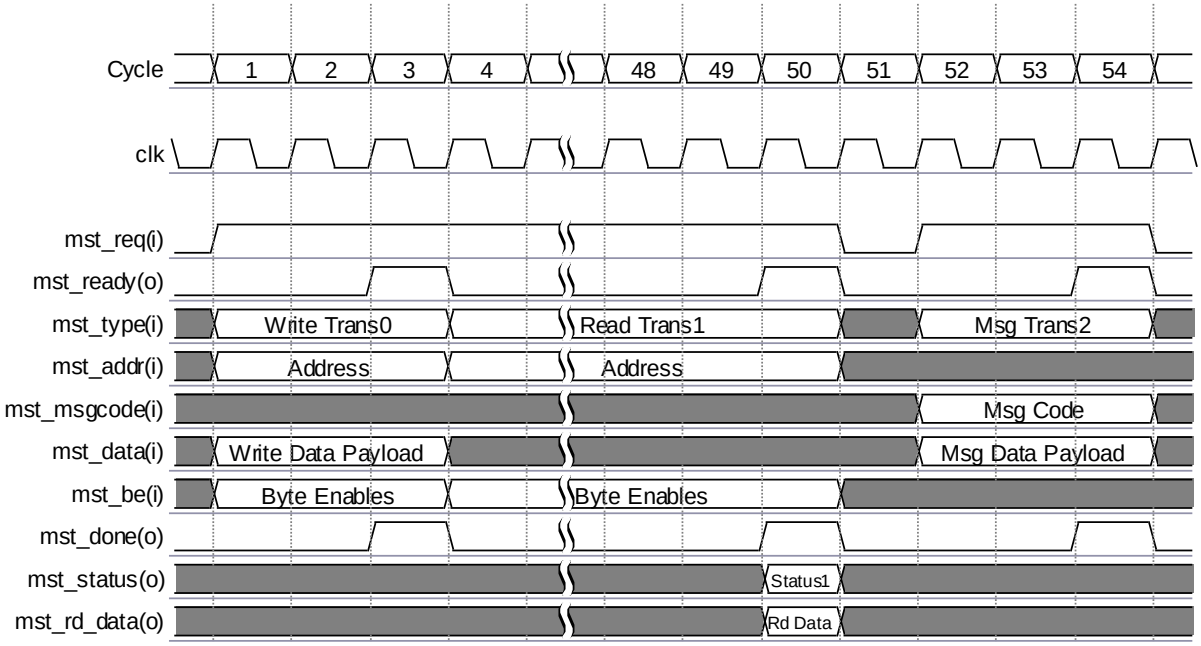


Figure 0-1 Master Interface Example Transactions

A Master write request begins on cycle 1 as indicated by mst_req == 1, mst_type indicates a write request, mst_data is the data to write, and mst_be are the transaction byte enables. It is assumed that this is a memory write request, which does not require a completion, and the transaction thus completes on cycle 3 when mst_ready == mst_done == 1. Had this been an I/O or Configuration write request, then the transaction would not have completed until the corresponding completion completed as would have been indicated by mst_done == 1 with mst_status == completion status.

Immediately following the master memory write a read request is seen to be pending when mst_req remains asserted, mst_type is changed to indicate a read request, and mst_be is updated on cycle 4. The read request is issued to the remote device and the status and data from the resulting completion is assumed to be returned on cycle 50 when mst_done == 1. On the same clock cycle, mst_status and mst_rd_data indicate the transaction status and returned read data payload.

On cycle 51, the master interface is idle as indicated by mst_req == 0.

On cycle 52, a Message with Data Payload transaction starts. This follows the same timing as the previously illustrated memory write, but uses the mst_msgcode and does not use mst_addr or mst_be inputs. A Message without Data Payload has identical timing but does not use the mst_data input.

13 Configuration Space

The Configuration Space is implemented by the PCI Express Core. Please see the relevant PCI Express Core User Guide for details. Note, however, that the following differences apply:

- 0x13-0x10 : Base Address Register locations 0 : The DMA Back-End Core requires that a 64KByte 32-bit not-prefetchable memory address Base Address Register be implemented at these locations for the Register Interface.

14 Migrating from Previous DMA Back-End Versions

Starting with version 4.0 of this User Guide, very significant enhancements and changes have been made to the DMA Back-End source code implementation:

- Packet DMA operation was added
  - DMA Register, Descriptors, and theory of operation are significant different than Block DMA
  - Since Packet DMA is new there are no migration issues
- Maximum frequency of operation substantially increased
  - The DMA Engine s were substantially recoded to increase the frequency of operation of the DMA Back-End and to enable a much higher level of overlapping for increased DMA efficiency and to be able to support high-performance Packet DMA operation.
- Block DMA Back-End operation is mostly the same as for prior versions. The primary difference is that the Card to System DMA Engines no longer support the “Sequence” and “Continue” functionality of previous versions and no longer handle byte alignment issues for the user. DMA byte alignment issues now need to be handled by the user logic. The DMA Back-End reference design illustrates how this is accomplished.
- Relevant DMA Back-End versions:
  - 1.5.xx and higher DMA Back-End versions implement the functionality in this User Guide.
  - 1.4.xx and prior DMA Back-End versions adhere to the DMA Back-End Core User Guide 3.11

Please contact Northwest Logic if additional information about migrating from a DMA Back-End version 1.4.xx and earlier to DMA Back-End version 1.5xx and later is required.

15 Packet DMA Reference Design

A synthesizable reference design is provided for Packet DMA Engines to enable robust simulation and hardware verification of the Packet DMA Streaming Interface and DMA Engines independent of the user design. An additional goal is to provide enough flexibility over the packet generation to enable users to model their bandwidth requirements and test whether the targeted systems can sustain the desired throughputs. Different systems have better performance than others primarily due to larger PCI Express payload sizes and more efficient or higher-bandwidth system SDRAM controllers.

The reference design provides Packet Generators to simulate providing DMA packet data for Card to System Packet DMA Engines and Packet Checkers to consume DMA packet data provided by System to Card Packet DMA Engines.

15.1 Packet Streaming Interface Port Descriptions

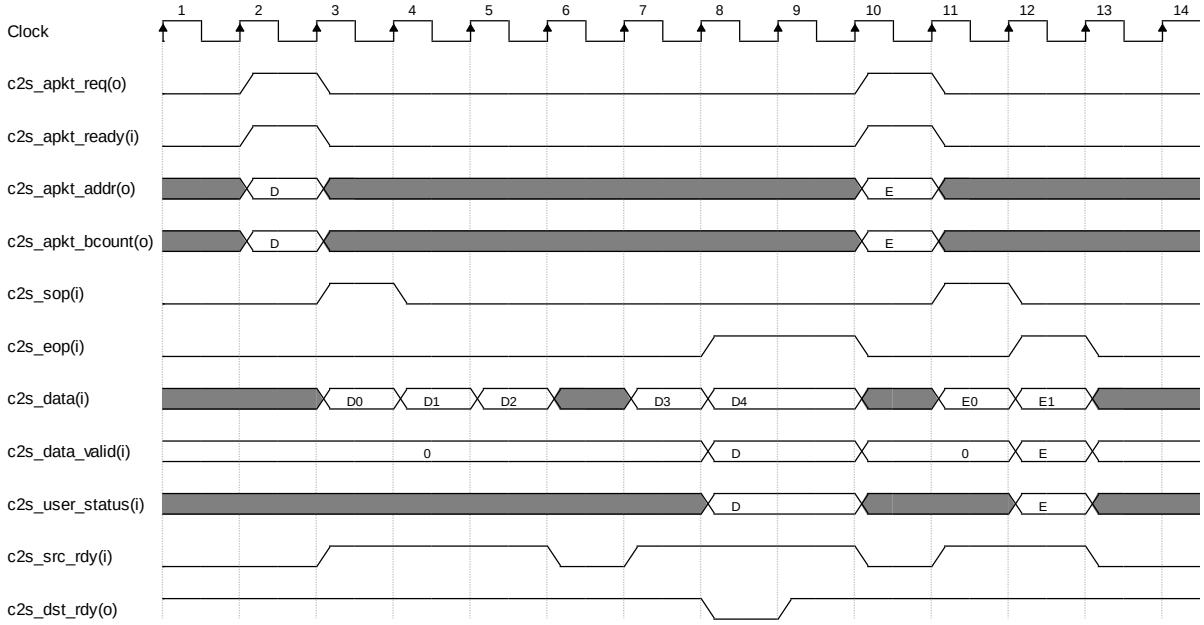
Table 15-1 Card to System Packet Streaming Interface Ports

PORT	TYPE	DESCRIPTION
c2s[C-1:0]_user_status[63:0]	Input	For communicating application specific information between the user's logic and software. The value of c2sX_user_status is accepted on the Packet DMA Interface when c2sX_src_rdy == 1 and c2sX_dst_rdy == 1 and c2sX_eop == 1. The value captured on c2sX_user_status is copied into the Descriptor status of the same Descriptor that contains the EOP == 1 Descriptor status.
c2s[C-1:0]_sop	Input	Indicates Start-of-packet. c2sX_sop must be set when c2s_src_rdy == 1 and the first data word for the packet is present on c2sX_data.
c2s[C-1:0]_eop	Input	Indicates End-of-packet. c2sX_eop must be set when c2s_src_rdy == 1 and the final data word for the packet is present on c2sX_data.
c2s[C-1:0]_data[D-1:0]	Input	Valid data must be provided on c2sX_data with each clock that c2sX_src_ready == 1. Data transfers when c2sX_src_rdy == 1 and c2sX_dst_rdy == 1. c2sX_data width D is the same as the Core Data Width.
c2s[C-1:0]_data_valid[R-1:0]	Input	When c2sX_src_rdy == 1 and c2sX_eop == 1, c2sX_valid must correctly indicate the number of bytes of c2sX_data that are used (part of the packet) in the final packet data word; 0 == all bytes are used; at all other times c2sX_valid must be 0. R is 4 for 128-bit CORE_DATA_WIDTH; 3 for 64-bit CORE_DATA_WIDTH, and 2 for 32-bit CORE_DATA_WIDTH cores
c2s[C-1:0]_src_rdy	Input	Source ready. When c2sX_src_rdy == 1, s2cX_sop, c2sX_eop, c2sX_data, and c2sX_user_status must all be valid. Data transfers when c2sX_src_rdy == 1 and c2sX_dst_rdy == 1.
c2s[C-1:0]_dst_rdy	Output	Destination ready. Data transfers when c2sX_src_rdy == 1 and c2sX_dst_rdy == 1.
c2s[C-1:0]_abort	Output	Set when the DMA Engine register DMA_Reset_Request has been set indicating that software is requesting that all pending DMA operations be aborted and the user design should prepare for reset. Cleared by an assertion of c2s[C-1:0]_abort_ack.

c2s[C-1:0]_abort_ack	Input	Set when the user design is ready to allow a reset to occur. Should be set for one clock when c2s[C-1:0]_abort is asserted and the user is ready to let a reset occur.
c2s[C-1:0]_user_rst_n	Output	Active low reset signal. Should be used for all logic that interfaces to this port. c2s[C-1:0]_user_rst_n is asserted whenever the corresponding DMA Engine has been reset.
c2s[C-1:0]_apkt_req	Output	Addressed packet transaction request. When c2sX_apkt_req == 1, c2sX_apkt_addr and c2sX_apkt_bcount will indicate the address and length of the next memory read transaction.
c2s[C-1:0]_apkt_ready	Input	Address Packet transaction ready. Transactions are accepted when c2sX_apkt_req==1 and c2sX_apkt_ready == 1. If addressed packets are not used, this signal should be tied high.
c2s[C-1:0]_apkt_addr	Output	Addressed packet read address. Indicates the start address for the next packet.
c2s[C-1:0]_apkt_bcount	Output	Addressed packet byte count. Indicates the total number of bytes to be read to complete the next descriptor. The length of a packet which spans several descriptors will be the sum of multiple c2sX_apkt_bcount values from successive requests. The last one will be marked by c2sX_apkt_eop.
c2s[C-1:0]_apkt_eop	Output	Addressed packet end of packet. Indicates that the data in this request ends the packet. If c2sX_apkt_eop == 0, then the packet continues into the next descriptor.

15.2 Card to System Packet Streaming Interface Timing Diagrams

The following diagram illustrates the acceptance of two packets “D” and “E”. Note that c2s_valid is only non-zero when c2s_eop is asserted. Note that and c2s_user_status is only valid when c2s_eop is asserted. Note that ports are only advanced when c2s_src_rdy == 1 and c2s_dst_rdy == 1 indicating that both the source and destination are ready.



15.3 System to Card Packet Streaming Interface Ports

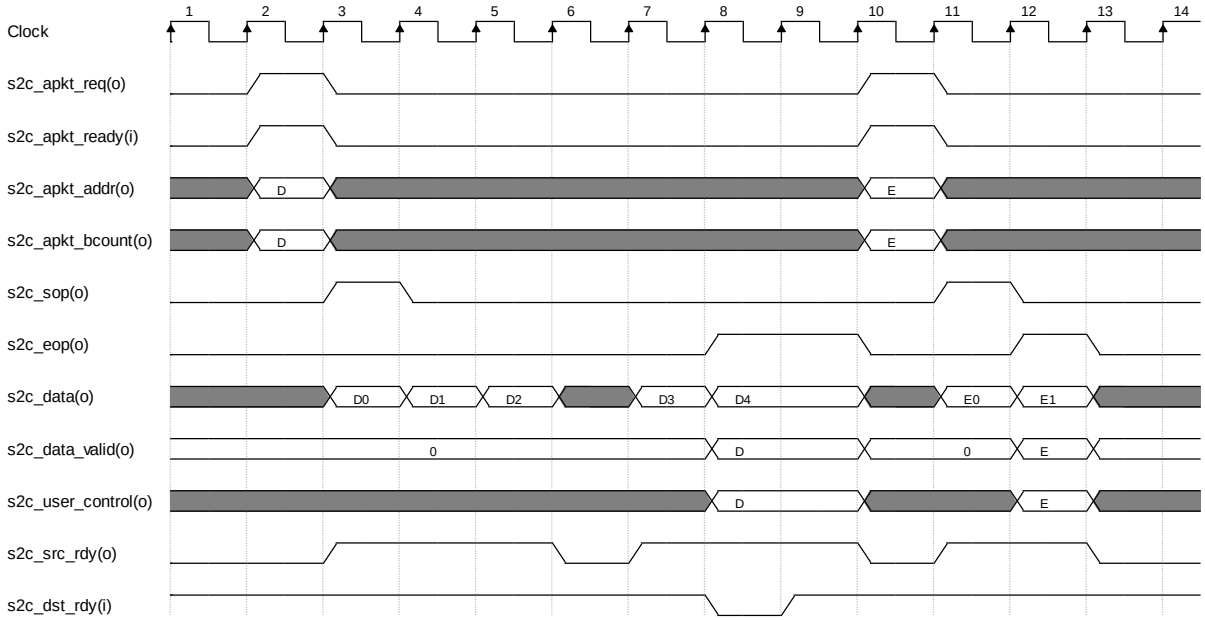
Table 15-2 System to Card Packet Streaming Interface Ports

PORT	TYPE	DESCRIPTION
s2c[C-1:0]_user_control[63:0]	Output	For communicating application specific information between software and the user's logic. The value of s2cX_user_control is valid when s2cX_src_rdy == 1 and s2cX_sop == 1. The value provided on s2cX_user_control is copied from the current Descriptor's S2CDescUserControl field.
s2c[C-1:0]_sop	Output	Indicates start-of-packet. s2cX_sop is set when s2c_src_rdy == 1 and the first data word for the packet is present on s2cX_data.
s2c[C-1:0]_eop	Output	Indicates end-of-packet. s2cX_eop is set when s2c_src_rdy == 1 and the final data word for the packet is present on s2cX_data.
s2c[C-1:0]_err	Output	When set indicates that the DMA Engine detected an error (read request was not completed). The current packet contents are not valid.
s2c[C-1:0]_data[D-1:0]	Output	s2cX_data is valid when s2cX_src_ready == 1. Data transfers when s2cX_src_rdy == 1 and s2cX_dst_rdy == 1. s2cX_data width D is the same as the Core Data Width.
s2c[C-1:0]_data_valid[R-1:0]	Output	When s2cX_src_rdy == 1 and s2cX_eop == 1, s2cX_valid indicates the number of bytes of s2cX_data that are used (part of the packet) in the final packet data word; 0 == all bytes are used; at all other times s2cX_valid is 0. R is 4 for 128-bit CORE_DATA_WIDTH; 3 for 64-bit CORE_DATA_WIDTH, and 2 for 32-bit CORE_DATA_WIDTH cores
s2c[C-1:0]_src_rdy	Output	Source ready. When s2cX_src_rdy == 1, s2cX_sop, s2cX_eop, s2cX_data, and s2cX_user_control are all valid. Data transfers when s2cX_src_rdy == 1 and s2cX_dst_rdy == 1.
s2c[C-1:0]_dst_rdy	Input	Destination ready. Data transfers when s2cX_src_rdy == 1 and s2cX_dst_rdy == 1.
s2c[C-1:0]_abort	Output	Set when the DMA Engine register DMA_Reset_Request has been set indicating that software is requesting that all pending DMA operations be aborted and the user design should prepare for reset. When s2c[C-1:0]_abort is asserted, the user should allow all packet operations on this port to complete relatively quickly. Packet data should be accepted and processed when possible and should be discarded if it cannot be accepted for a significant period. s2c[C-1:0]_abort is cleared in response to an assertion of s2c[C-1:0]_abort_ack.
s2c[C-1:0]_abort_ack	Input	Set for one clock in response to s2c[C-1:0]_abort being asserted when the user design is ready to allow a reset to occur.

s2c[C-1:0]_user_rst_n	Output	Active low reset signal. Should be used for all logic that interfaces to this port. s2c[C-1:0]_user_rst_n is asserted whenever the corresponding DMA Engine has been reset.
S2c[C-1:0]_apkt_req	Output	Addressed packet transaction request. When c2sX_apkt_req == 1, c2sX_apkt_addr and c2sX_apkt_bcount will indicate the address and length of the next memory write transaction.
s2c[C-1:0]_apkt_ready	Input	Address Packet transaction ready. Transactions are accepted when c2sX_apkt_req==1 and c2sX_apkt_ready == 1. If addressed packets are not used, this signal should be tied high.
s2c[C-1:0]_apkt_addr	Output	Addressed packet write address. Indicates the start address for the next block of data. Data within the same packet will always be written to a continuous range of addresses. A single packet can be broken up into many blocks.
s2c[C-1:0]_apkt_bcount	Output	Addressed packet byte count. Indicates the total number of bytes in the next block of data.

15.4 System to Card Packet Streaming Interface Timing Diagrams

The following diagram illustrates the acceptance of two packets “D” and “E”. Note that s2c_valid is only non-zero when s2c_eop is asserted. Note that and s2c_user_control is only valid when s2c_sop is asserted. Note that data values are only advanced when s2c_src_rdy == 1 and s2c_dst_rdy == 1 indicating that both the source and destination are ready.



15.5 Card to System Packet Generator

A hardware-synthesizable Packet Generator is implemented in the reference design to enable generation of known packets to emulate user source packet data. The Packet Generator enables the Card to System Packet DMA operation to be robustly checked by the test bench environment and also by software in a hardware implementation. The Packet Generator directly connects to the DMA Back-End Register Interface to implement its register set and to its associated Card to System Packet DMA Engine Packet Streaming Interface to provide source packet data.

Each Card to System Packet DMA Engine has its own associated Packet Generator. The Packet Generators are controlled via BAR0 registers that are located at the BAR0 offset of the DMA Engine that they are associated with plus 0xA000.

The process for testing DMA operation with the Packet Generator is as follows:

- Setup Packet Generator and DMA Engine
- Start Packet Generator
- Start DMA Engine
- Process completed Descriptors and verify that SOP, EOP, packet length, packet payload, and packet C2SDescUserStatus contents match the values programmed into the Packet Generator

The Packet Generator provides the following programmability (see Table 15-3 for details):

- Number of packets generated – infinite or limited number
- Packet rate – generate different data rate packet streams
- Packet length – cycles through up to 4 pre-loaded values
- Auto incrementing packet payload data – known pattern that can be checked
- Auto incrementing of UserStatus – known pattern that can be checked

The packet generator implements 32-bit data patterns, so for example, a 128-bit core data width core has 4 (128/32) pattern values per word. A packet generated with Data_Pattern == PAT_INC_DWORD and Data_Seed = 32'h0 for a 128-bit core generates the following packet data output:

- {32'h3, 32'h2, 32'h1, 32'h0}
- {32'h7, 32'h6, 32'h5, 32'h4}
- {32'hb, 32'ha, 32'h9, 32'h8}
- ..

If a pattern continues from packet to packet then the pattern starting value for a new packet is one pattern value greater than the last 32-bits of the prior packet that had any portion of the 32-bits valid. For example, for a 64-bit core, packet data for two packets using Data_Pattern == PAT_INC_BYTE, Data_Seed == 32'h03020100 and Continuous_Data_Pattern == 1 would be generated as follows:

- Packet0 (length 10 bytes):
  - {32'h07060504, 32'h03020100}; data_valid = 0
  - {32'hxxxxxxxx, 32'hxxxx0908} ; data_valid = 2
- Packet1 (length 8 bytes)
  - {32'h13121110, 32'h0f0e0d0c}; data_valid = 0

Table 15-3 Packet Generator Register Block

Byte Address	Description
0x3 - 0x0	Control[31:0] – Read/Write Bit[0] – Enable; set to have the Packet Generator create and forward packets to the associated Packet DMA Engine Interface. This bit clears when the final packet generation is complete or on the first end of packet after this register has been written to 0. A read of Enable == 1 indicates that the Packet Generator is busy with a prior request. Bit[1] – Loopback_Enable; 1==Loopback; 0==Normal operation. The Packet Generator is capable of looping back packets (transmitting received packets) when paired with a Packet Checker which also has its Loopback_Enable register enabled. In the standard Northwest Logic reference design, each DMA Engine is paired with an engine of the opposite direction with the same engine number. For instance S2C0 is paired with C2S0, S2C1 with C2S1, etc. If a pair of engines does not exist (for example unequal numbers of C2S and S2C engines are included in the reference design) then Loopback_Enable may not be used. Loopback_Enable and Enable are mutually exclusive. It is illegal to have both asserted. Bit[2] – Packet Source Select, 0==Packet Generator; 1==Other source. This bit controls a select line output from the packet generator which indicates whether the streaming packet interface should receive packets from the packet generator. In the reference design, this bit is used to select between the packet generator and the memory reader used with addressed packet mode. Bits[3] - Reserved Bits[5:4] – Packet_Table_Entries[1:0] – Highest index to use in the Packet_Length_Table. Entries are used to determine packet length in increasing numerical order from 0 to Packet_Table_Entries. When the index == Packet_Table_Entires, the index is reset to 0. Bits[7:6] - Reserved Bits[10:8] – Data_Pattern[2:0]; 32-bit data pattern to use for c2sX_data; the pattern starts with the 32-bit Data_Seed value; the next value in the pattern is according to Data_Pattern using unsigned math. PAT_LFSR is provided to generate pseudo random sequences. • 0 - PAT_CONSTANT; next = current • 1 - PAT_INC_BYTE; for each BYTE next == current + 4 • 2 - PAT_LFSR; for each DWORD next =={current[30:0], ~(^(current[31:0] & 32'b1000_0000_0010_0000_0000_0100_0000_0011))}; • 3 - PAT_INC_DWORD; for each DWORD next == current + 1 • 7:4 - Reserved Bit[11] – Continuous_Data_Pattern • 1 == Continue the data pattern across packet boundaries • 0 == Start the data pattern with Data_Seed for each packet start Bits[14:12] – User_Status_Pattern[2:0]; pattern to use for c2sX_ user_status; the pattern starts with the User_Status_Seed value; the next value in the pattern is according to User_Status_Pattern using unsigned math. PAT_LFSR is provided to generate pseudo random sequences. • 0 - PAT_CONSTANT; next = current • 1 - PAT_INC_BYTE; for each BYTE next == current + 4 • 2 - PAT_LFSR; for each DWORD next =={current[30:0], ~(^(current[31:0] & 32'b1000_0000_0010_0000_0000_0100_0000_0011))}; • 3 - PAT_INC_DWORD; for each DWORD next == current + 1 • 7:4 - Reserved Bit[15] – Continuous_User_Status_Pattern • 1 == Continue the user_status pattern across packet boundaries • 0 == Assign User_Status_Seed to user_status for all generated packets Bits[23:16] – Active_Clocks[7:0]; see Packet Rate Control - Section 15.7 Bits[31:24] – Inactive_Clocks[7:0]; see Packet Rate Control - Section 15.7

Table 15-3 Packet Generator Register Block (cont)

Byte Address	Description
0x7 - 0x4	NumPackets[31:0] – Read/Write Bits[31:0] – Number of packets to generate. On the rising edge of Enable, NumPackets is sampled to determine how many packets to generate before stopping. NumPackets == 0 is a special case and indicates infinite packets. In this case, packets continue to be generated as long as Enable == 1.
0x0B - 0x08	Data_Seed[31:0] – Read/Write Data pattern starting seed. Seed is used for the first DWORD (32-bits) of the pattern. All subsequent pattern words are determined by the pattern specified in Data_Pattern.
0x0F - 0x0C	User_Status_Seed[31:0] – Read/Write User status starting seed. Seed is used for the first DWORD (32-bits) of the pattern. All subsequent pattern words are determined by the pattern specified in User_Status_Pattern.
0x1F - 0x10	Reserved
0x2F-0x20	Packet Length Table – Read/Write Array of four 32-bit values used to control generated packet length: • Bits[31:20] Reserved • Bits[19:0] Packet_Length[19:0] in bytes On the rising edge of Enable, array_index is reset to 0 and the first entry in the table is used to determine the first generated packet's length. After each packet generation completes, if array_index == Packet_Table_Entries then the index is reset to 0, otherwise array_index is incremented by 1 to the next entry in the table.
0xFF - 0x30	Reserved

15.6 System to Card Packet Checker

A hardware-synthesizable Packet Checker is implemented in the reference design to consume and enable checking of system-generated System to Card DMA Packets. The Packet Checker directly connects to the DMA Back-End Register Interface to implement its register set and to its associated System to Card Packet DMA Engine Packet Streaming Interface to consume packet data.

Each System to Card Packet DMA Engine has its own associated Packet Checker. The Packet Checkers are controlled via BAR0 registers that are located at the BAR0 offset of the DMA Engine that they are associated with plus 0xA000.

The process for testing DMA operation with the Packet Checker is as follows:

- Setup Packet Checker and DMA Engine
- Start Packet Checker
- Start DMA Engine
- Provide Descriptors to the DMA Engine with the packet sizes and contents matching the values loaded into the Packet Checker
- The Packet Checker verifies that SOP, EOP, packet length, packet payload, and packet S2CDescUserControl contents match the values programmed into the Packet Checker registers; errors are noted in the registers for read-back, and for simulation, to the simulation log

The Packet Checker provides the following programmability (see Table 15-4 for details):

- Number of packets checked – infinite or limited number
- Packet rate – model consumption of packets at different data rates
- Packet length – cycles through up to 4 pre-loaded expected values
- Auto incrementing packet payload data – check against known pattern
- Auto incrementing of UserControl – check against known pattern

The packet checker implements 32-bit data patterns, so for example, a 128-bit core data width core has 4 (128/32) pattern values per word. A packet checked with Data_Pattern == PAT_INC_DWORD and Data_Seed = 32'h0 for a 128-bit core expects the following packet input data:

- {32'h3, 32'h2, 32'h1, 32'h0}
- {32'h7, 32'h6, 32'h5, 32'h4}
- {32'hb, 32'ha, 32'h9, 32'h8}
- ..

If a pattern continues from packet to packet then the pattern starting value for a new packet is one pattern value greater than the last 32-bits of the prior packet that had any portion of the 32-bits valid. For example, for a 64-bit core, packet data for two packets using Data_Pattern == PAT_INC_BYTE, Data_Seed == 32'h03020100 and Continuous_Data_Pattern == 1 would be generated as follows:

- Packet0 (length 10 bytes):
  - {32'h07060504, 32'h03020100}; data_valid = 0
  - {32'hxxxxxxxx, 32'hxxxx0908} ; data_valid = 2
- Packet1 (length 8 bytes)
  - {32'h13121110, 32'h0f0e0d0c}; data_valid = 0

Table 15-4 Packet Checker Register Block

Byte Address	Description
0x3 - 0x0	Control[31:0] – Read/Write Bit[0] – Enable; set to have the Packet Checker consume and check packets from the associated Packet DMA Engine Interface. This bit clears when the final packet check is complete or on the first end of packet after this register has been written to 0. A read of Enable == 1 indicates that the Packet Checker is busy with a prior request. Bit[1] – Loopback_Enable; 1==Loopback; 0==Normal operation. The Packet Checker is capable of looping back packets (transmitting received packets) when paired with a Packet Generator which also has its Loopback_Enable register enabled. In the standard Northwest Logic reference design, each DMA Engine is paired with an engine of the opposite direction with the same engine number. For instance S2C0 is paired with C2S0, S2C1 with C2S1, etc. If a pair of engines does not exist (for example unequal numbers of C2S and S2C engines are included in the reference design) then Loopback_Enable may not be used. Loopback_Enable and Enable are mutually exclusive. It is illegal to have both asserted. Bit[2] - Packet Destination Select, 0==Packet Checker; 1==Other destination. This bit controls a select line output from the packet checker which indicates whether the packet checker should receive packets from the streaming packet interface. In the reference design, this bit is used to select between the packet checker and the memory writer used with addressed packet mode. Bits[3] - Reserved Bits[5:4] – Packet_Table_Entries[1:0] – Highest index to use in the Packet_Length_Table. Entries are used to determine packet length in increasing numerical order from 0 to Packet_Table_Entries. When the index == Packet_Table_Entires, the index is reset to 0. Bits[7:6] - Reserved Bits[10:8] – Data_Pattern[2:0]; pattern to use to check s2cX_data; the pattern starts with the Data_Seed value; the next value in the pattern is according to Data_Pattern using unsigned math. PAT_LFSR is provided to generate pseudo random sequences. • 0 - PAT_CONSTANT; next = current • 1 - PAT_INC_BYTE; for each BYTE next == current + 4 • 2 - PAT_LFSR; for each DWORD next ={current[30:0], ~(^(current[31:0] & 32'b1000_0000_0010_0000_0000_0100_0000_0011))}; • 3 - PAT_INC_DWORD; for each DWORD next == current + 1 • 7:4 - Reserved Bit[11] – Continuous_Data_Pattern • 1 == Continue the data pattern across packet boundaries • 0 == Start the data pattern with Data_Seed for each packet start Bits[14:12] – User_Control_Pattern[2:0]; pattern to use to check s2cX_ user_control; the pattern starts with the User_Control_Seed value; the next value in the pattern is according to User_Control_Pattern using unsigned math. PAT_LFSR is provided to generate pseudo random sequences. • 0 - PAT_CONSTANT; next = current • 1 - PAT_INC_BYTE; for each BYTE next == current + 4 • 2 - PAT_LFSR; for each DWORD next ={current[30:0], ~(^(current[31:0] & 32'b1000_0000_0010_0000_0000_0100_0000_0011))}; • 3 - PAT_INC_DWORD; for each DWORD next == current + 1 • 7:4 - Reserved Bit[15] – Continuous_User_Control_Pattern • 1 == Continue the user_control pattern across packet boundaries • 0 == Check against User_Control_Seed for all generated packets Bits[23:16] – Active_Clocks[7:0]; see Packet Rate Control - Section 15.7 Bits[31:24] – Inactive_Clocks[7:0]; see Packet Rate Control - Section 15.7

Table 15-4 Packet Checker Register Block (cont)

Byte Address	Description
0x7 - 0x4	NumPackets[31:0] – Read/Write Bits[31:0] – Number of packets to check. On the rising edge of Enable, NumPackets is sampled to determine how many packets to check before stopping. NumPackets == 0 is a special case and indicates infinite packets. In this case, packets continue to be checked as long as Enable == 1.
0x0B - 0x08	Data_Seed[31:0] – Read/Write Data pattern starting seed. Seed is used for the first DWORD (32-bits) of the pattern. All subsequent pattern words are determined by the pattern specified in Data_Pattern.
0x0F - 0x0C	User_Control_Seed[31:0] – Read/Write User control starting seed. Seed is used for the first DWORD (32-bits) of the pattern. All subsequent pattern words are determined by the pattern specified in User_Control_Pattern.
0x10	Error[31:0] – Bit[7] - err_clear is Write Only; others Read Only; Bits[5:0] set and remain set (until cleared by a write of 1 to err_clear) on the first detected error. Errors detected by the Packet Checker are logged in this register. • Bit[0] – err_sop : Set on a SOP error (1 when 0 expected; 0 when 1 expected) • Bit[1] – err_eop : Set on a EOP error (1 when 0 expected; 0 when 1 expected) • Bit[2] – err_cpl : Set when the DMA Engine detected and signaled an error on the "err" port (a DMA read request was unsuccessful – usually due to a bad Descriptor) • Bit[3] – err_data : Set when a packet data error is detected • Bit[4] – err_data_valid : Set when receiving the expected eop location and data_valid was not the expected value • Bit[5] - err_user_control : Set when receiving the expected sop location and user_control was not the expected value • Bits[6] – Reserved • Bit[7] – err_clear : Software writes a 1 to err_clear to clear this register of all logged errors. • Bits[31:8] – err_ctr : Incremented by 1 for every clock that one or more error events (of the types logged in Bits[5:0]) occurred. err_ctr saturates at 24'hffffff.
0x1F - 0x14	Reserved
0x2F-0x20	Packet Length Table – Read/Write Array of four 32-bit expected packet length values: • Bits[31:20] Reserved • Bits[19:0] Packet_Length[19:0] in bytes On the rising edge of Enable, array_index is reset to 0 and the first entry in the table is used to check the first consumed packet's length. After each packet generation completes, if array_index == Packet_Table_Entries then the index is reset to 0, otherwise array_index is incremented by 1 to the next entry in the table.
0xFF - 0x30	Reserved

15.7 Packet Rate Control

The Packet Generator and Checker each implement the following registers to control packet data rate:

- Active_Clocks[7:0]
- Inactive_Clocks[7:0]

These two register specify the desired data ready time (Active_Clocks) and data not ready time (Inactive_Clocks) for c2sX_src_rdy for the Packet Generator and s2cX_dst_rdy for the Packet Checker. The data ready rate is computed independent of whether the c2sX_dst_rdy/s2cX_src_rdy signal is asserted by the interface connected to the Packet Generator/Checker, so the interface connected to the Packet Generator/Checker must always be ready in order to achieve the desired date rate.

The packet data rate assuming that the other interface is always ready is:

- [Active_Clocks/(Active_Clocks+Inactive_Clocks)] * (max possible data rate)
    - max possible data rate depends upon CORE_DATA_WIDTH and clock freq:
        - Ex: 128-bit, 250 MHz => 16 bytes / 4nS == 4 billion bytes/second
        - Ex: 64-bit, 125 MHz => 8 bytes / 8nS == 1 billion bytes/second
    - max possible data rate is achieved when Active_Clocks != 0 and Inactive_Clocks == 0: [1/(1+0)] == 1/1 * max_possible_data_rate
    - min possible rate is achieved when Active_Clocks == 1 and Inactive_Clocks = 255: [1/(1+255) == 1/256 * max possible data rate.
    - Some other examples (A == Active_Clocks; I == Inactive_Clocks):
        - A=1, I=1: [1/(1+1)] = 1/2
        - A=1, I=3: [1/(1+3)] = 1/4
        - A=3, I=2: [3/(3+2)] = 3/5
        - A=5, I=7: [5/(5+7)] = 5/12
        - A=253; I=3; [253/(253+3)] = 253/256
        - A=3; I=253; [3/(3+253)] = 3/256

16 Addressed Card Memory DMA Reference Design

A synthesizable reference design is provided to illustrate use of the Block DMA Engine or Packet DMA Engine (using addressed packet mode) for high-bandwidth DMA transfer to and from system memory and card memory. If licensing a supported version of Northwest Logic's SDRAM Controller and Multi-Port Front End cores in conjunction with the DMA Back-End, then the reference design includes these cores connected to the DMA Back-End in the manner illustrated in Figure 2-1. Otherwise the reference design, that is delivered, is the same as Figure 2-1 except a multi-ported internal SRAM is instantiated in place of the SDRAM Controller and Multi-Port Front End. Northwest Logic test software is setup to accept either of these configurations. The main observable difference to software is that the internal SRAM version is much less deep than the SDRAM version.

The reference design includes FIFOs between the DMA Back-End DMA Interfaces and the Multi-Port Front End interfaces of the SDRAM controller. Many users will find that they can greatly simplify their own user design by utilizing these FIFOs. Generally the DMA side of the FIFOs and the FIFOs themselves can be used as-is so that only the SDRAM-side of the FIFOs need to be modified to perform the desired connection to user logic.

17 RAM Usage

This core contains some functions which require on-chip RAM arrays (also referred to as embedded memory or block RAM). The standard RTL code delivered by Northwest Logic implements these RAM arrays using generic Verilog or VHDL array constructs.

For FPGA targets, FPGA synthesis tools typically will automatically convert these arrays into the appropriate FPGA block RAMs or if necessary, Northwest Logic will pre-configure the IP Core with the proper RAM instantiations prior to delivery. In either case the designer does not need to be involved in this process. The FPGA designer only needs to ensure that there is enough block RAM available in their target part.

For ASIC targets, the designer needs to be actively involved in configuring the RAM arrays. ASIC flows generally do not support automatic generation of RAM arrays so the designer must first configure and create the RAM arrays using a memory generator tool and then substitute these in place of the inferred memory blocks provided in the IP core. To verify the actual size of the RAM, please follow the procedure outlined in the ASIC Integration Guide available from Northwest Logic. If there are any questions on the RAM configuration or handling of the substitution, please contact Northwest Logic.

An example report you will get when the simulation starts is as follows:

```
# tb_top.dut.pcie_complete_core_vc1.dma_back_end_pkt.s2c0_dma_engine_pkt.s2c_tx_pkt.cpl_reorder_
queue_pkt.caddr_ram: RAM Instance using ADDR_WIDTH=   4, DATA_WIDTH=  64, FAST_READ=   0
```

As you can see, this report shows the instance in the design hierarchy where the RAM is located, as well as the information on how the RAM is sized.

18 Speed & Size

For speed & size information, see the target device family document: ***Speed & Size Overview***

This document can be found on Northwest Logic's secure website. Request access to the secure website by sending an email to **nwl@nwlogic.com**.

19 Free Evaluation Core

To receive a free evaluation core, follow the following steps:

1. Request access to Northwest Logic's secure website by sending an e-mail to **nwl@nwlogic.com**
2. Log into the secure website section of Northwest Logic's website at **www.nwlogic.com**
3. Download the Evaluation Request Form for the core you are interested in
4. Fill in Table 3-2
5. E-mail the Evaluation Request Form to **nwl@nwlogic.com**
6. You should receive the Evaluation Core within 24 hours

20 For More Information

For more information including licensing options, pricing and the latest version of this document:

- Visit our website at **www.nwlogic.com**
- Send an e-mail to **nwl@nwlogic.com**
- Call us at **503-533-5800 x309**

Northwest Logic is located at:

Address: 1100 NW Compton Drive, Suite 100
Beaverton, Oregon 97006
United States

Phone: 503-533-5800

Fax: 503-533-5900