NORTHWEST LOGIC

Expresso Testbench For PCI Express®

User Guide

To The Point Solutions

Version 2.02

NORTHWEST LOGIC

Table Of Contents

Table Of Contents

1 Revision History

This section tracks revisions made to this document by version number

Revision	Date	Changes
2.01	02/09/2009	Added revision history section to the document.
2.02	08/20/2010	Further clarified which portions of testbench are delivered as source code and which as encrypted source code.

2 Overview

The Expresso Testbench provides a simulation test environment for developing PCI Express logic designs utilizing the Northwest Logic Expresso family of cores for PCI Express®.

The Expresso Testbench includes a Root Complex Bus Functional Model (pcie_bfm), test scripts (ref_design_ts), and logging functionality as illustrated in Figure 2-1. The Expresso Testbench transmits and receives PCI Express traffic from the Device Under Test (DUT) and enables a comprehensive full-system simulation.

The Expresso Testbench, except for pcie_model, is delivered as Verilog source code for all licensees. pcie_model, which is Northwest Logic's Expresso Core configured for Root Port operation, is delivered as source code if the equivalent Expresso Core with Root Port support has been licensed as source code. Otherwise, pcie_model is delivered as encrypted source code.

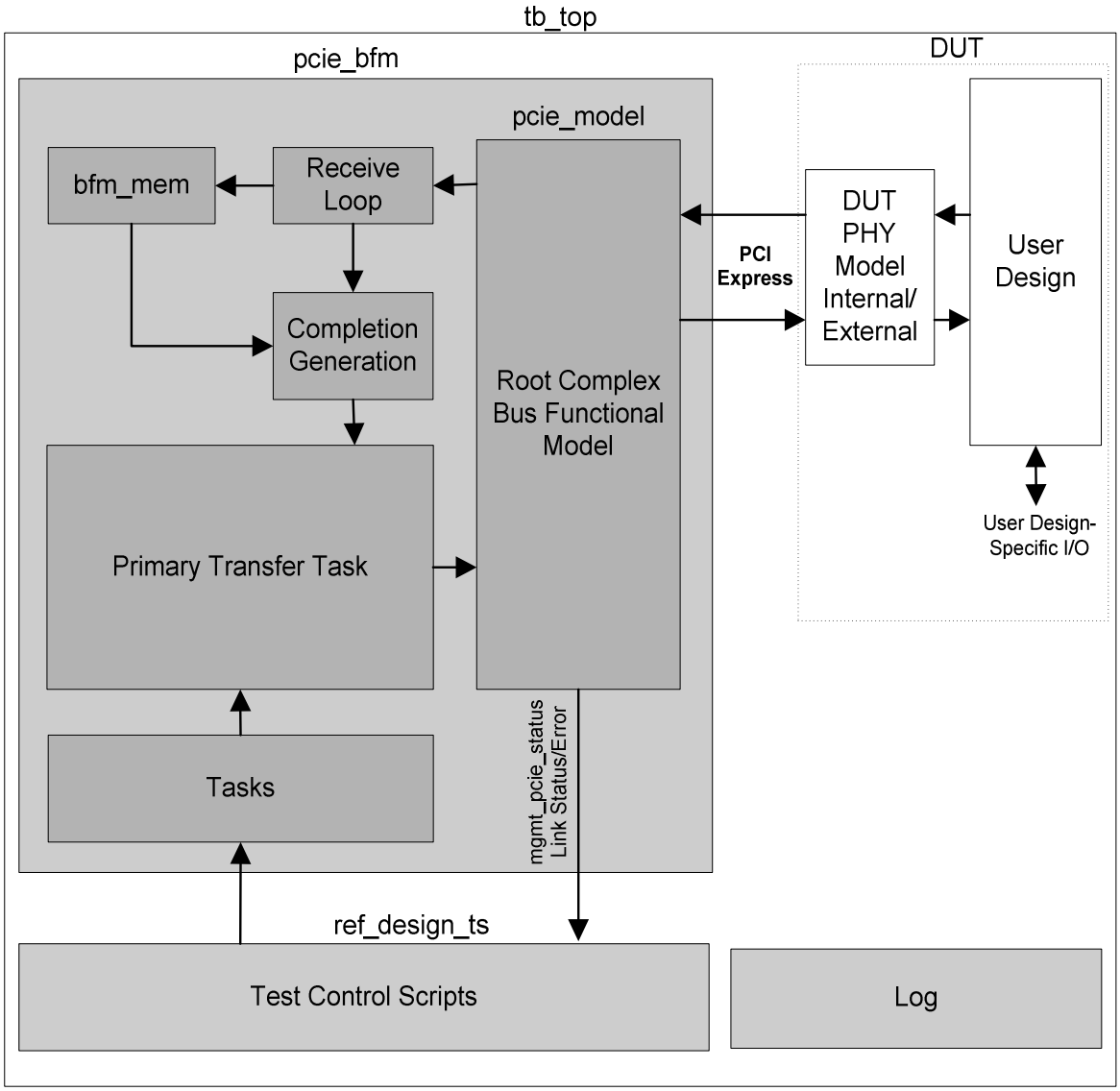

Figure 2-1 Typical Test Environment

The Expresso Testbench is available in Verilog 2001. VHDL customers may also use the Expresso Testbench by co-simulating with a simulator which supports Verilog and VHDL.

Primary Testbench Components

- tb_top
  - Example top level testbench
  - The user customizes this file by adding their DUT and associated DUT I/O models
- pcie_bfm
  - Delivered as Verilog source code except pcie_model (see below)
  - Provides high level tasks to master PCI Express requests to the DUT
    - Tasks are very simple to use and require very little PCI Express knowledge
    - Can generate virtually any PCI Express request
    - Enables full control over payload contents including auto generation of pattern data
    - Auto checks the payload of read request completions for expected data
    - BFM task calls mirror the way software accesses PCI Express devices; use is intuitive
  - Autonomously processes requests received from the DUT
    - Received Memory Write Requests are automatically consumed and the data payload is written into BFM target memory, bfm_mem, at the addressed location
    - Received Memory Read Requests are automatically consumed and the corresponding completions are automatically generated and transmitted to the DUT using the payload data from BFM target memory, bfm_mem, at the addressed location
    - DUT master requests interact with the BFM in the same manner that they would interact with the system; bfm_mem memory contents are updated by DUT writes and can be retrieved with DUT reads
  - Detailed link status and error information is available on mgmt_pcie_status from pcie_model
    - See Northwest Logic Expresso Core User Guide for detailed information on this port
    - Error information from this port is also output to the Log
- pcie_model
  - Northwest Logic Expresso Core configured for Root Port operation including a behavioral PCI Express PHY
  - Delivered as Verilog source code for Root Port source code licensees of Northwest Logic Expresso Core for PCI Express
  - Delivered as encrypted source code for Endpoint and netlist licensees of Northwest Logic Expresso Core and stand-alone licensees of Expresso Testbench
- ref_design_ts
  - Example test control scripts illustrating BFM operation
    - Scans the Type 0 Configuration Space of the DUT and sets up all discovered memory, I/O, and Expansion ROM windows, and enables the DUT
    - Stores BAR address locations for DUT and BFM in arrays for easy referencing
    - Numerous example scripts for verifying DUT target resources (Memory, IO BARs)
    - Numerous example scripts for verifying DMA operation for DUTs which contain the Northwest Logic DMA Back-End Core
  - This file is modified by the user to customize/add tests to verify the features of the DUT
    - Simple to use
    - Many examples to start from
- Log
  - Transaction Layer traffic is optionally logged to standard output and/or file I/O
  - Assertion monitor optionally records error and status information to the Log
  - User can easily add information to the Log
- DUT
  - User Device Under Test including the relevant PCI Express PHY model, DUT, and DUT I/O models
  - Generally includes Northwest Logic's Expresso and/or DMA Back-End Cores

Users generally do not need to modify any of the Expresso Testbench files other than tb_top to instantiate their DUT and DUT-specific I/O models and ref_design_ts to develop custom tests to verify the DUT.

3 Tasks for Generating Traffic

The following tasks are defined in pcie_bfm for users to generate PCI Express traffic. All tasks listed below (except xfer_action) may be called simultaneously from multiple initial or fork/join blocks. The tasks automatically handle arbitration to share transmit resources.

3.1 xfer

xfer is the main PCI Express transmit task and can be used to transmit nearly any type of packet. Additional tasks are defined in the following sections which use task xfer to generate requests but which are more intuitive to use.

<table>
<tr><th>Port</th><th>Direction</th><th>Size</th><th>Description</th></tr>
<tr><td>tc</td><td>Input</td><td>[2:0]</td><td>Traffic Class; Use 000 for general PCI Express traffic</td></tr>
<tr><td>fmt_and_type</td><td>Input</td><td>[6:0]</td><td>PCI Express Command Type {fmt, type}. Defined per PCI Express Specification:
<table>
<tr><th>TLP Type</th><th>Fmt [1:0]</th><th>Type [4:0]</th><th>Description</th></tr>
<tr><td>Mrd32</td><td>00</td><td>00000</td><td>Mem Read (32-bit)</td></tr>
<tr><td>Mrd64</td><td>01</td><td>00000</td><td>Mem Read (64-bit)</td></tr>
<tr><td>MrdLk32</td><td>00</td><td>00001</td><td>Mem Read Lock(32-bit)</td></tr>
<tr><td>MrdLk64</td><td>01</td><td>00001</td><td>Mem Read Lock (64-bit)</td></tr>
<tr><td>MWr32</td><td>10</td><td>00000</td><td>Mem Write (32-bit)</td></tr>
<tr><td>MWr64</td><td>11</td><td>00000</td><td>Mem Write (64-bit)</td></tr>
<tr><td>IORd</td><td>00</td><td>00010</td><td>I/O Read</td></tr>
<tr><td>IOWr</td><td>10</td><td>00010</td><td>I/O Write</td></tr>
<tr><td>CfgRd0</td><td>00</td><td>00100</td><td>Type 0 Configuration Read</td></tr>
<tr><td>CfgWr0</td><td>10</td><td>00100</td><td>Type 0 Configuration Write</td></tr>
<tr><td>CfgRd1</td><td>00</td><td>00101</td><td>Type 1 Configuration Read</td></tr>
<tr><td>CfgWr1</td><td>10</td><td>00101</td><td>Type 1 Configuration Write</td></tr>
<tr><td>Msg</td><td>01</td><td>10rrr</td><td>Message</td></tr>
<tr><td>MsgD</td><td>11</td><td>10rrr</td><td>Message with Data</td></tr>
<tr><td>Cpl</td><td>00</td><td>01010</td><td>Completion</td></tr>
<tr><td>CplD</td><td>10</td><td>01010</td><td>Completion with Data</td></tr>
<tr><td>CplLk</td><td>00</td><td>01011</td><td>Completion Locked</td></tr>
<tr><td>CplDLk</td><td>10</td><td>01011</td><td>Completion Locked with Data</td></tr>
</table></td></tr>
<tr><td>length</td><td>Input</td><td>[9:0]</td><td>Length of payload in DWORDs (0 == 1024)</td></tr>
<tr><td>first_dw_be</td><td>Input</td><td>[3:0]</td><td>First DWORD Byte Enables; for Msg/MsgD == MsgCode[3:0]</td></tr>
<tr><td>last_dw_be</td><td>Input</td><td>[3:0]</td><td>Last DWORD Byte Enables; for Msg/MsgD == MsgCode[7:4]</td></tr>
<tr><td>addr</td><td>Input</td><td>[63:0]</td><td>Starting address; for Msg/MsgD == Message Bytes 15 to 8; addr[63:32] must be 0 unless a 64-bit fmt_and_type is used; Note: addr[1:0] is reserved for most packet types and the first_dw_be and last_dw_be ports are used to specify non-DWORD aligned starting and ending addresses</td></tr>
<tr><td>req_id</td><td>Input</td><td>[15:0]</td><td>Requester ID (only used for BFM-generated completions)</td></tr>
<tr><td>tag</td><td>Input</td><td>[7:0]</td><td>Requester Tag (only used for BFM-generated completions)</td></tr>
<tr><td>payload</td><td>InOut</td><td>[32767:0]</td><td>Transaction payload to use for write requests; expected transaction payload for read requests; payload is a bit array supporting up to the maximum payload size of 4KBytes. payload[31:0] is the first DWORD, payload[63:32] is 2nd DWORD, etc.</td></tr>
<tr><td>check_data</td><td>Input</td><td>[0]</td><td>For read requests only: Set to check completion payload data against payload when the completions for this request are received; Clear to not check completion payload data</td></tr>
<tr><td>no_wait_for_cpl</td><td>Input</td><td>[0]</td><td>For non-posted requests only: Set to wait for all completions to complete before returning from the task call. Clear to not wait. If set, then when the task call completes, the completion data received for the transaction is output on the payload port. Test sequences which just need the completion data checked against the expected data do not need to wait. Test sequences which need the actual completion data received to proceed, must set this bit.</td></tr>
</table>

Task xfer may be called simultaneously from multiple initial and/or fork/join blocks and arbitrates for the shared resource, task xfer_action which actually carries out the requested packet transmission(s). Task xfer_action may only be called by task xfer. Since xfer_action is not directly callable by the user, it is not documented here, although its ports are the same as task xfer. Task xfer also uses two semaphore tasks get_xfer_key and put_xfer_key which also may not be called by the user and aid in the arbitration.

For requests that contain payload data (such as Memory Write Requests), the xfer task uses the payload data from the payload port. payload must be initialized with the desired payload data and must not contain any undefined values. Any undefined "X" or "Z" data bits will keep the PCI Express link from working due to an "X" or "Z" causing the cumulative-in-time CRC and scrambling functions of the PCI Express protocol to accumulate to X and thus fail to work properly.

The BFM automatically generates any required completions for received DUT non-posted requests. The BFM calls the xfer task to accomplish these completions. However, in this case, the xfer task uses the addressed locations in array bfm_mem, which is the same memory that DUT write requests are written into, as the completion payload data. Thus, the DUT write and reads to/from the BFM bfm_mem in the same manner as if the BFM were a regular target device. DUT writes update bfm_mem and DUT reads return the data from bfm_mem.

3.2 xferb

xferb has the same ports as task xfer with the exception that the length and address are byte values and the first and last byte enable fields are automatically generated.

<table>
<tr><th>Port</th><th>Direction</th><th>Size</th><th>Description</th></tr>
<tr><td>tc</td><td>Input</td><td>[2:0]</td><td>Traffic Class; Use 000 for general PCI Express traffic</td></tr>
<tr><td>fmt_and_type</td><td>Input</td><td>[6:0]</td><td>PCI Express Command Type {fmt, type}. Defined per PCI Express Specification:
<table>
<tr><th>TLP Type</th><th>Fmt [1:0]</th><th>Type [4:0]</th><th>Description</th></tr>
<tr><td>Mrd32</td><td>00</td><td>00000</td><td>Mem Read (32-bit)</td></tr>
<tr><td>Mrd64</td><td>01</td><td>00000</td><td>Mem Read (64-bit)</td></tr>
<tr><td>MrdLk32</td><td>00</td><td>00001</td><td>Mem Read Lock(32-bit)</td></tr>
<tr><td>MrdLk64</td><td>01</td><td>00001</td><td>Mem Read Lock (64-bit)</td></tr>
<tr><td>MWr32</td><td>10</td><td>00000</td><td>Mem Write (32-bit)</td></tr>
<tr><td>MWr64</td><td>11</td><td>00000</td><td>Mem Write (64-bit)</td></tr>
<tr><td>IORd</td><td>00</td><td>00010</td><td>I/O Read</td></tr>
<tr><td>IOWr</td><td>10</td><td>00010</td><td>I/O Write</td></tr>
<tr><td>CfgRd0</td><td>00</td><td>00100</td><td>Type 0 Configuration Read</td></tr>
<tr><td>CfgWr0</td><td>10</td><td>00100</td><td>Type 0 Configuration Write</td></tr>
<tr><td>CfgRd1</td><td>00</td><td>00101</td><td>Type 1 Configuration Read</td></tr>
<tr><td>CfgWr1</td><td>10</td><td>00101</td><td>Type 1 Configuration Write</td></tr>
<tr><td>Msg</td><td>01</td><td>10rrr</td><td>Message</td></tr>
<tr><td>MsgD</td><td>11</td><td>10rrr</td><td>Message with Data</td></tr>
<tr><td>Cpl</td><td>00</td><td>01010</td><td>Completion</td></tr>
<tr><td>CplD</td><td>10</td><td>01010</td><td>Completion with Data</td></tr>
<tr><td>CplLk</td><td>00</td><td>01011</td><td>Completion Locked</td></tr>
<tr><td>CplDLk</td><td>10</td><td>01011</td><td>Completion Locked with Data</td></tr>
</table></td></tr>
<tr><td>blength</td><td>Input</td><td>[11:0]</td><td>Length of payload in bytes (0 == 4096)</td></tr>
<tr><td>addr</td><td>Input</td><td>[63:0]</td><td>Starting byte address; blength and addr are translated into the DWORD aligned address and first_dw_be and last_dw_be ports required by the xfer task.</td></tr>
<tr><td>req_id</td><td>Input</td><td>[15:0]</td><td>Requester ID (only used for BFM-generated completions)</td></tr>
<tr><td>tag</td><td>Input</td><td>[7:0]</td><td>Requester Tag (only used for BFM-generated completions)</td></tr>
<tr><td>payload</td><td>InOut</td><td>[32676:0]</td><td>Transaction payload to use for write requests; expected transaction payload for read requests; payload is a bit array supporting up to the maximum payload size of 4KBytes. payload[31:0] is the first DWORD, payload[63:32] is 2nd DWORD, etc.</td></tr>
<tr><td>check_data</td><td>Input</td><td>[0]</td><td>For read requests only: Set to check completion payload data against payload when the completions for this request are received; Clear to not check completion payload data</td></tr>
<tr><td>no_wait_for_cpl</td><td>Input</td><td>[0]</td><td>For non-posted requests only: Set to wait for all completions to complete before returning from the task call. Clear to not wait. If set, then when the task call completes, the completion data received for the transaction is output on the payload port. Test sequences which just need the completion data checked against the expected data do not need to wait. Test sequences which need the actual completion data received to proceed, must set this bit.</td></tr>
</table>

3.3 config_read

config_read performs a single Type 0 Configuration Space read and waits for the resulting completion. The read data from the subsequent completion is not available. If the read data is needed then use task config_read_bdf instead.

Port	Direction	Size	Description
addr	Input	[11:0]	Configuration register address
be	Input	[3:0]	Byte Enables

3.4 config_write

config_write performs a single Type 0 Configuration Space write and waits for the resulting completion.

Port	Direction	Size	Description
addr	Input	[11:0]	Configuration register address
data	Input	[31:0]	Write data
be	Input	[3:0]	Byte Enables

3.5 config_read_bdf

config_read_bdf performs a single Type 0 Configuration Space read and waits for the resulting completion. The read data from the subsequent completion is output on read_data.

Port	Direction	Size	Description
addr	Input	[11:0]	Configuration register address
be	Input	[3:0]	Byte Enables
dut_bdf	Input	[15:0]	Bus[7:0], Device[4:0], Function[2:0] to address with the read
read_data	Output	[31:0]	Returned read data

3.6 config_write_bdf

config_write performs a single Type 0 Configuration Space write and waits for the resulting completion.

Port	Direction	Size	Description
addr	Input	[11:0]	Configuration register address
data	Input	[31:0]	Write data
be	Input	[3:0]	Byte Enables
dut_bdf	Input	[15:0]	Bus[7:0], Device[4:0], Function[2:0] to address with the write

3.7 transmit_msg

Task transmit_msg transmits the desired message packet. Care should be exercised to only transmit valid message types. Please see the PCI Express Specification for a list of messages which are defined.

Port	Direction	Size	Description
tc	Input	[1:0]	Traffic Class
msgd_msg_n	Input	[0]	1 == Transmit message with payload; 0 == Transmit message without payload
msg_routing	Input	[2:0]	Message routing type • 000 – Routed to Root Complex • 001 – Routed by Address • 010 – Routed by ID • 011 – Broadcast from Root Complex • 100 – Local – Terminate at Receiver • 101 – Gathered and routed to Root Complex • 110 – Reserved • 111 – Reserved
msg_code	Input	[7:0]	Message Code; see PCI Express Specification for options
msg_info	Input	[63:0]	Contents of the message; varies according to Message Type; see PCI Express Specification for options
msg_data	Input	[31:0]	Message payload (only used if msgd_msg_n == 1)

3.8 mem_write_dword_addr32

mem_write_dword_addr32 performs a single 32-bit memory write using a 32-bit address.

Port	Direction	Size	Description
tc	Input	[2:0]	Traffic Class
addr	Input	[31:0]	Address
data	Input	[31:0]	Write data
be	Input	[3:0]	Byte Enables

3.9 mem_read_dword_addr32

mem_read_dword_addr32 performs a single 32-bit memory read using a 32-bit address and waits for the resulting completion.

Port	Direction	Size	Description
tc	Input	[2:0]	Traffic Class
addr	Input	[31:0]	Address
expect_data	Input	[31:0]	Expected Data
be	Input	[3:0]	Byte Enables; all enabled bytes are checked and errors are logged if an enabled expect_data byte does not match an enabled read_data byte
read_data	Output	[31:0]	Returned read data

3.10 mem_write_dword

mem_write_dword performs a single 32-bit memory write and supports both 32-bit and 64-bit addresses.

Port	Direction	Size	Description
tc	Input	[2:0]	Traffic Class
addr	Input	[63:0]	Address
data	Input	[31:0]	Write data
be	Input	[3:0]	Byte Enables

3.11 mem_read_dword

mem_read_dword performs a single 32-bit memory read and supports both 32-bit and 64-bit addresses and waits for the resulting completion.

Port	Direction	Size	Description
tc	Input	[2:0]	Traffic Class
addr	Input	[63:0]	Address
expect_data	Input	[31:0]	Expected Data
be	Input	[3:0]	Byte Enables; all enabled bytes are checked and errors are logged if an enabled expect_data byte does not match an enabled read_data byte
read_data	Output	[31:0]	Returned read data

3.12 mem_read_dword_fast

mem_read_dword_fast performs a single 32-bit memory read, supports both 32-bit and 64-bit addresses, does not wait for the resulting completion, and optionally checks the completion data.

Port	Direction	Size	Description
tc	Input	[2:0]	Traffic Class
addr	Input	[63:0]	Address
expect_data	Input	[31:0]	Expected Data
check_data	Input	[0]	Set to check the completion data against expect_data

3.13 io_write_dword

io_write_dword performs a single 32-bit I/O write and waits for the resulting completion.

Port	Direction	Size	Description
tc	Input	[2:0]	Traffic Class
addr	Input	[63:0]	Address; addr[63:32] must be 0x0 since I/O space is only 32-bit
data	Input	[31:0]	Write data
be	Input	[3:0]	Byte Enables

3.14 io_read_dword

io_read_dword performs a single 32-bit I/O read and waits for the resulting completion.

Port	Direction	Size	Description
tc	Input	[2:0]	Traffic Class
addr	Input	[63:0]	Address; addr[63:32] must be 0x0 since I/O space is only 32-bit
expect_data	Input	[31:0]	Expected Data
be	Input	[3:0]	Byte Enables; all enabled bytes are checked and errors are logged if an enabled expect_data byte does not match an enabled read_data byte
read_data	Output	[31:0]	Returned read data

3.15 mem_write_burst

mem_write_burst generates a burst memory write request using the payload data contained in the payload port. payload should not contain any undefined values that could cause the link to transmit an X as this will be fatal for the link due to the cumulative nature of PCI Express CRC, scrambling, etc. functions.

Port	Direction	Size	Description
tc	Input	[2:0]	Traffic Class
addr	Input	[63:0]	Starting Address
length	Input	[9:0]	Length in DWORDS; 0 == 1024
first_dw_be	Input	[3:0]	First DWORD Byte Enables
last_dw_be	Input	[3:0]	Last DWORD Byte Enables
payload	Input	[32676:0]	Transaction payload to use for the write request; payload is a bit array supporting up to the maximum payload size of 4KBytes. payload[31:0] is the first DWORD, payload[63:32] is 2nd DWORD, etc.

3.16 mem_write_burst_pattern

mem_write_burst_pattern generates a burst memory write request using the specified data pattern for payload data.

Port	Direction	Size	Description
tc	Input	[2:0]	Traffic Class
addr	Input	[63:0]	Starting Address
length	Input	[9:0]	Length in DWORDS; 0 == 1024
first_dw_be	Input	[3:0]	First DWORD Byte Enables
last_dw_be	Input	[3:0]	Last DWORD Byte Enables
start_data	Input	[31:0]	Pattern seed value; the first DWORD of the pattern
pattern	Input	[31:0]	Pattern type to use; uses the same constants defined in task get_pattern

3.17 mem_read_burst

mem_read_burst generates a burst memory read request and waits for the resulting completion(s). If check_data == 1, then the received data is checked against payload and any violations are noted in the log.

Port	Direction	Size	Description
tc	Input	[2:0]	Traffic Class
addr	Input	[63:0]	Starting Address
length	Input	[9:0]	Length in DWORDS; 0 == 1024
check_data	Input	[0]	If set, the task checks the corresponding completions agains payload and logs any errors.
first_dw_be	Input	[3:0]	First DWORD Byte Enables
last_dw_be	Input	[3:0]	Last DWORD Byte Enables
payload	InOut	[32676:0]	Expected transaction payload for the read requests; on task completion contains the received completion data for the request; payload is a bit array supporting up to the maximum payload size of 4KBytes. payload[31:0] is the first DWORD, payload[63:32] is 2nd DWORD, etc.

3.18 mem_read_burst_pattern

mem_read_burst generates a burst memory read request and waits for the resulting completion(s). If check_data == 1, then the received data is checked against the specified pattern and any violations are noted in the log.

Port	Direction	Size	Description
tc	Input	[2:0]	Traffic Class
addr	Input	[63:0]	Starting Address
length	Input	[9:0]	Length in DWORDS; 0 == 1024
check_data	Input	[0]	If set, the task checks the corresponding completions agains payload and logs any errors.
first_dw_be	Input	[3:0]	First DWORD Byte Enables
last_dw_be	Input	[3:0]	Last DWORD Byte Enables
start_data	Input	[31:0]	Pattern seed value; the first DWORD of the pattern
pattern	Input	[31:0]	Pattern type to use; uses the same constants defined in task get_pattern

3.19 do_multi_dma_g3

Task do_multi_dma_g3 sets up and executes a DMA operation on a DUT which contains the Northwest Logic DMA Back-End. The DUT must contain a version of the Northwest Logic DMA Back-End for this task to work as intended.

Please see the DMA Back-End User Guide for DMA operation details. Reading this document is required to understand the use of the do_multi_dma_g3 ports.

Port	Direction	Size	Description
reg_com_bar	Input	[63:0]	Memory address where the DMA Common register block is located (DUT BAR0 + 0x4000)
reg_dma_bar	Input	[63:0]	Memory address where the desired DMA engine's registers are located; System to Card engines are located at DUT BAR0 + 0x0, +0x100, etc. Card to System DMA Engines are located at DUT BAR0 + 0x2000, +0x2100, etc.
system_addr	Input	[63:0]	Starting physical byte address of DMA in system memory (bfm_mem)
card_addr	Input	[63:0]	Starting physical byte address of DMA in card memory (DUT DMA memory); Note: DMA applications which connect to FIFOs rather than addressable memory, such as SDRAM, generally ignore card_addr and should set card_addr =0.
bcount	Input	[31:0]	DMA total transfer byte count
desc_ptr	Input	[63:0]	System address location where the first DMA Descriptor should be located
done_wait	Input	[0]	1 == Wait for the DMA completion interrupt before continuing. 0 == Do not wait for the DMA Interrupt before continuing. If done_wait == 1, then the task does not finish execution until the final DMA completion interrupt is received. In this case all DMA memory transfers will have completed to bfm_mem or the DUT memory and the transferred data can be checked for validity. If done_wait == 0, then the task will return as soon as the DMA is started and it will be unknown how long the DMA will take to complete which will make checking the DMA data validity more difficult
num_desc	Output	[31:0]	The number of Descriptors that the do_multi_dma_g3 task created in order to transfer bcount bytes of data in the DMA.

The do_multi_dma_g3 task is designed to emulate the way a DMA occurs in a typical operating system (OS). A DMA operation transfers data between the system memory (bfm_mem) and card memory (DUT DMA memory). Normally an application passes a large buffer to the DUT's driver and requests that the buffer either be transferred to the card or the card's data be transferred to the buffer. Before the driver can use the application's buffer, the driver must request that the OS map the application's buffer from its (typically) virtual memory representation into physical memory that can be accessed via DMA. The OS normally maps the application buffer into 4KByte pages that are often not contiguous in physical memory. Each of these memory fragments is translated into a DMA Descriptor. The DMA Descriptors are chained together in a linked list and the address of the head of the chain is written into the DMA engine registers.

The do_multi_dma_g3 task emulates the mapping of the application to physical memory by breaking bcount along 4KByte system_addr boundaries. One Descriptor is created and placed in bfm_mem starting at the address specified by desc_ptr for each such memory fragment and all of the resulting Descriptors are chained together into one DMA chain. num_desc returns the number of Descriptors which were created in case this information is needed. Each Descriptor requires 32 bytes.

After the Descriptors are created and placed into bfm_mem for the DUT DMA engine to fetch, the DMA is started by making the appropriate writes to the DMA engine registers.

Task do_multi_dma_g3 always uses legacy INTA as the expected interrupt source.

DUTs utilizing the DMA Back-End may support up to 4 Card to System and up to 4 System to Card engines. The example ref_design_ts.v file delivered as part of the Expresso Testbench with a DMA Back-End Core delivery contains example logic which scans DUT BAR0 to determine how many engines of each type are present and then sets up an array of this information for convenient reference when making do_multi_dma_g3 task calls.

4 Support Tasks

The following tasks do not directly generate PCI Express traffic and are provided to facilitate use of the main traffic generating tasks.

4.1 init_payload

init_payload generates the desired payload data using the desired data pattern.

Port	Direction	Size	Description
start_data	Input	[31:0]	Pattern seed value; the first DWORD of the pattern
pattern	Input	[31:0]	Pattern type to use; uses the same constants defined in task get_pattern
Payload	Output	[32767:0]	Generated payload data; payload is a bit array supporting up to the maximum payload size of 4KBytes. payload[31:0] is the first DWORD, payload[63:32] is 2nd DWORD, etc.

4.2 init_bfm_mem

init_bfm_mem initializes a range of bfm_mem using the specified data pattern. All bytes are initialized between start_addr and start_addr+byte_count-1. bfm_mem is used to return completion data to DUT requests. init_bfm mem is useful for initializing bfm_mem with a known pattern which can be checked by the DUT after a master read.

Port	Direction	Size	Description
start_addr	Input	[63:0]	Starting byte address
byte_count	Input	[31:0]	Length in bytes; 0 == invalid
start_data	Input	[31:0]	Pattern seed value; the first DWORD of the pattern
pattern	Input	[31:0]	Pattern type to use; uses the same constants defined in task get_pattern

4.3 get_pattern

Task get_pattern produces the next value in a pattern from the current data value and a pattern type and is used to auto generate a known sequence of data. get_pattern is used by tasks which operate on patterns.

Port	Direction	Size	Description
pattern	Input	[31:0]	Pattern to use to generate next from curr
curr	Input	[31:0]	Current data value in pattern
next	Output	[31:0]	Next data value in pattern

The following constants are defined for the pattern port:

Name	Value	Description	Examples
PAT_CONSTANT	0	next = curr	curr, curr, curr, ..
PAT_ONES	1	next = 0xFFFFFFFF	0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, ..
PAT_ZEROS	2	next = 0x00000000	0x00000000, 0x00000000, 0x00000000, ..
PAT_INC_NIB	3	Increment (Mod16) each nibble next[3:0] = curr[3:0] + 8 next[7:4] = curr[7:4] + 8 next[11 8] = curr[11:8] + 8 next[15:12] = curr[15:12] + 8 next[19:16] = curr[19:16] + 8 next[23:20] = curr[23:20] + 8 next[27:24] = curr[27:24] + 8 next[31:28] = curr[31:28] + 8	0x76543210, 0xfedcba98, 0x76543210, ..
PAT_INC_BYTE	4	Increment (Mod256) each byte next[7:0] = curr[7:0] + 4 next[15:8] = curr[15:8] + 4 next[23:16] = curr[23:16] + 4 next[31:24] = curr[31:24] + 4	0x03020100, 0x07060504, 0x0b0a0908, ..
PAT_INC_WORD	5	Increment (Mod16K) each 16-bits next[15:0] = curr[15:0] + 2 next[31:16] = curr[31:16] + 2	0x00010000, 0x00030002, 0x00050004, ..
PAT_INC_DWORD	6	Increment (Mod 4G) each 32-bits next[31:0] = curr[31:0] + 1	0x00000000, 0x00000001, 0x00000002, ..
PAT_L_SHIFT	7	Left shift 1 bit position next[31:0] = {curr[30:0], 0}	0x40000000, 0x80000000, 0x00000000, ..
PAT_R_SHIFT	8	Right shift 1 bit position next[31:0] = {0, curr[31:1]}	0x00000002, 0x00000001, 0x00000000, ..
PAT_L_ROT	9	Left rotate 1 bit position next[31:0] = {curr[30:0], curr[31]}	0x40000000, 0x80000000, 0x00000001, ..
PAT_R_ROT	10	Right rotate 1 bit position next[31:0] = {curr[0], curr[31:1]}	0x00000002, 0x00000001, 0x80000000, ..

4.4 Task display_hdr

The Testbench includes the option to log transmitted and received Transaction Layer packet information. By default the header information describing the log fields is printed once at the beginning of the simulation. Task display_hdr is called by the user to repeat the header information and is included to enhance the readability of the log output. Example log output for task display_hdr follows:

```
#                                                    F C    Cp        C     M
#                                                  L i p Cp B   C  C  f  C  s
#                                                  a r l L  C   f  f  g  f  g
#                                                  s s S A  o   g  g  F  g  C
#                                                  t t t d  u   B  D  u  R  o
#                                                  B B a d  n   u  e  n  e  d
#                                                  E E t r  t   s  v  c  g  e
#
#      CMD  Tg RqID CpID Len  Addr64   Addr32  L F S LA Bct Bs Dv F Reg MC  Time(nS)
#     ----- -- ---- ---- --- -------- -------- - - - -- --- -- -- - --- -- ----------
```

4.5 Task display_short_hdr

Task display_short_hdr is the same as task display_hdr but prints only the short notation. Example log output for task display_short_hdr follows:

```
#      CMD  Tg RqID CpID Len  Addr64   Addr32  L F S LA Bct Bs Dv F Reg MC  Time(nS)
#     ----- -- ---- ---- --- -------- -------- - - - -- --- -- -- - --- -- ----------
```

4.6 Task inc_errors

Task inc_errors is called for every error which should be logged. Each call to inc_errors will increase the error count by 1.

4.7 Task clear_errors

Task clear_errors resets the error counter to 0 (no errors).

4.8 Task report_status

Displays to the Log the number of errors received as tracked by error counter or a message that all tests passed if the error counter == 0. Generally clear_errors is called before the test sequence is started, inc_errors is called once for each detected error, and report_status is called once at the end of the test sequence to report the pass/fail status.

5 Responding to DUT Requests

The BFM automatically consumes DUT requests targeting BFM resources and generates any required completions.

DUT write requests are consumed and the payload data is written to bfm_mem at the address indicated in the packet.

DUT read requests are consumed and the required completions are automatically generated. Completion data is provided from bfm_mem at the address specified in the read request packet.

DUT write and read requests interact with bfm_mem in the same manner as a real target memory device. For example, write data can be observed to have been written correctly by reading the written locations.

When a DUT request is received that requires a completion, the request's tag and other information required for issuing completions are written into an array. An independent loop continuously monitors this array for uncompleted requests and closes the requests by generating the necessary completions. This process is done without any user intervention.

To emulate typical system read completion behavior, read requests crossing a 64 byte (RCB) address boundary are broken along 64 byte address boundaries into multiple completions.

6 DUT PHY Model

The Expresso Testbench interfaces to the DUT through PCI Express serial lanes. The DUT should include the appropriate PHY for PCI Express model for the PHY being used in/with the DUT.

7 Device Under Test (DUT)

The user's PCI Express device to be tested in the test environment is instantiated at the top level of the Testbench. Accesses to user memory/I/O/Configuration Space regions are made using task calls. The tasks have been written such that they execute in much the same manner as S/W executing on a microprocessor would appear on the PCI Express bus.

Using the Expresso Testbench, the user may easily and comprehensively test their device with a large variety of stimulus.

8 Transaction Logging

The Expresso Testbench supports logging Transaction Layer traffic to standard output and to a file. An example log is shown below. The fields follow the PCI Express Specification definitions for packet generation. The user is provided a handle to the log(s) and may insert their own log data as desired. The lines below starting with "# tb_tob.ref_design_ts" are examples of user-implemented logging information.

```
# tb_top.ref_design_ts : ######## PHYSICAL LAYER UP ########
# tb_top.ref_design_ts : ######## DATA LINK LAYER UP ########
# Transaction Layer packet display log:
#   * D - Downstream traffic (BFM Tx to DUT Rx)
#   * U - Upstream   traffic (DUT Tx to BFM Rx)
#   * For best viewing results, use a fixed width font
#
#                                                 F C    Cp        C     M
#                                               L i p Cp B   C  C  f  C  s
#                                               a r l L  C   f  f  g  f  g
#                                               s s S A  o   g  g  F  g  C
#                                               t t t d  u   B  D  u  R  o
#                                               B B a d  n   u  e  n  e  d
#                                               E E t r  t   s  v  c  g  e
#
#      CMD  Tg RqID CpID Len  Addr64   Addr32  L F S LA Bct Bs Dv F Reg MC  Time(nS)
#     ----- -- ---- ---- --- -------- -------- - - - -- --- -- -- - --- -- ----------
# tb_top.ref_design_ts : ** Configure Device Under Test **
# tb_top.ref_design_ts : Initialize Completer ID
# D : CfgWr 00 0080 ---- 001 -------- -------- 0 0 - -- --- 00 11 0 000 --       11464
# D :   Data == 00000000
# U : Cpl   00 0080 0088 000 -------- -------- - - 0 00 004 -- -- - --- --       13128
# tb_top.ref_design_ts : Read Vendor and Device ID
# D : CfgRd 01 0080 ---- 001 -------- -------- 0 f - -- --- 00 11 0 000 --       13144
# U : CplD  01 0080 0088 001 -------- -------- - - 0 00 004 -- -- - --- --       14840
# U :   Data (Tag 01) == e00119aa
# tb_top.ref_design_ts : Setup Base Address Registers
# D : CfgWr 02 0080 ---- 001 -------- -------- 0 f - -- --- 00 11 0 010 --       14840
# D :   Data == f0000000
# U : Cpl   02 0080 0088 000 -------- -------- - - 0 00 004 -- -- - --- --       16520
# D : CfgWr 03 0080 ---- 001 -------- -------- 0 f - -- --- 00 11 0 014 --       16536
# D :   Data == e0000000
# U : Cpl   03 0080 0088 000 -------- -------- - - 0 00 004 -- -- - --- --       18216
# D : CfgWr 04 0080 ---- 001 -------- -------- 0 f - -- --- 00 11 0 018 --       18232
# D :   Data == d0000000
# U : Cpl   04 0080 0088 000 -------- -------- - - 0 00 004 -- -- - --- --       19912
# D : CfgRd 05 0080 ---- 001 -------- -------- 0 f - -- --- 00 11 0 010 --       19928
# U : CplD  05 0080 0088 001 -------- -------- - - 0 00 004 -- -- - --- --       21624
# U :   Data (Tag 05) == f0000000
# D : CfgRd 06 0080 ---- 001 -------- -------- 0 f - -- --- 00 11 0 014 --       21624
# U : CplD  06 0080 0088 001 -------- -------- - - 0 00 004 -- -- - --- --       23320
# U :   Data (Tag 06) == e0000000
# D : CfgRd 07 0080 ---- 001 -------- -------- 0 f - -- --- 00 11 0 018 --       23320
# U : CplD  07 0080 0088 001 -------- -------- - - 0 00 004 -- -- - --- --       25032
# U :   Data (Tag 07) == d0000000
# tb_top.ref_design_ts : Enable SERR#, PERR#, Bus Master, Memory, and I/O
# D : CfgWr 08 0080 ---- 001 -------- -------- 0 3 - -- --- 00 11 0 004 --       25048
# D :   Data == 00000147
# U : Cpl   08 0080 0088 000 -------- -------- - - 0 00 004 -- -- - --- --       26728
# D : CfgRd 09 0080 ---- 001 -------- -------- 0 f - -- --- 00 11 0 004 --       26744
# U : CplD  09 0080 0088 001 -------- -------- - - 0 00 004 -- -- - --- --       28440
# U :   Data (Tag 09) == 00100147
```

9 Link Status/Error (mgmt_pcie_status)

pcie_model is a combination of Northwest Logic's Expresso Core configured for Root Port operation and a behavioral PCI Express PHY. pcie_model contains a wealth of PCI Express status and error information on mgmt_pcie_status. This information may optionally be included in the Log.

mgmt_pcie_status contains a wealth of diagnostic information including:

- Link Up/Down Status
- Error indicators
- PCI Express Credit information for both devices in the link
- Raw Physcial Layer parallel data (scrambled)
  - Physical Layer Transmit K, Data are provided to assist with Physcial Layer debug. Note: Physical Layer data except Ordered Sets and K Characters is scrambled per PCI Express Specification.
- Raw Data Link/Transaction Layer parallel data (unscrambled)
  - Data Link and Transaction Layer K, Data are provided to assist with Data Link and Transaction Layer debug. Data is unscrambled for the Data Link and Transaction Layer ports and is presented in the same format described in the Data Link and Transaction Layer portions of the PCI Express Specification.
- Signals to highlight interesting portions of the Data Link/Transaction Layer data such as
  - Highlight clock cycles containing TLPs and TLP type
  - Highlight clock cycles containing DLLP and DLLP type
- Key core state machines which can be informative for debugging problems with training

Please see the Expresso Core User Guide for details on the fields of the mgmt_pcie_status port.

10 Shortening Simulation Time

The Expresso Testbench contains two short cuts to reduce training time. pcie_model contains an input mgmt_short_sim which should be asserted to shorten training time. When mgmt_short_sim == 1, the core will use modified training as follows:

- Only 16 instead of the required 1024 TS1 sets will be required to be transmitted in Polling.Active prior to making progress
- All PCI Express training timeouts are reduced as follows:
  - 1 mS -> 10 uS (can't be too short since PHYs must be able to complete data rate changes)
  - 2 mS -> 5 uS
  - 12 mS -> 10 uS
  - 24 mS -> 20 uS
  - 48 mS -> 40 uS

mgtm_short_sim should normally be asserted to reduce training time. mgmt_short_sim may not be defined if the other PCI Express device in the link is not similarly modifying its training.

11 Licensing

The Expresso Testbench is delivered with Northwest Logic's Expresso and DMA Back-End Cores. It is not available on a stand-alone basis.

12 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